Universidad
Carlos III de Madrid

Departamento de Ingeniería de Sistemas y Automática

TRABAJO DE FIN DE GRADO

DESARROLLO DEL CONTROL POR CAN BUS DE UN CARRIL PARA LA SILLA DE RUEDAS DE ASIBOT

Autor: Roberto Gámez Pérez

Tutor: Juan Carlos González Víctores

Director: Alberto Jardón Huete

Leganés, septiembre de 2012

Título: DESARROLLO DEL CONTROL POR CAN BUS DE UN CARRIL PARA LA SILLA DE RUEDAS ASIBOT

Autor: Roberto Gámez Pérez

Tutor: Juan Carlos González Víctores

Director: Alberto Jardón Huete

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Trabajo de Fin de Grado el día __ de septiembre de 2012 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

SECRETARIO VOCAL PRESIDENTE

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

Quisiera agradecer a todas las personas que han hecho posible la realización de dicho proyecto, entre las cuales se encuentran:

- Agradecer a mi tutor Alberto Jardón Huete por haberme dado la oportunidad de poder realizar este proyecto, con el que he aumentado mi nivel formativo de manera considerable, además de todos los conocimientos que he adquirido.

- También me gustaría agradecer a Juan Carlos González Víctores, por haberme ayudado y guiado un proceso tan desconocido para mí. Además de aportarme gran cantidad de conocimientos acerca de la programación en C/C++. Ya que sin su gran ayuda no hubiera podido concluir dicho proyecto.

- Y por ultimo quiero agradecer a todas las personas que me ayudaron y apoyaron en estos meses desde que empecé el proyecto de final de carrera. Estas personas son mi familia y amigos, que sin su apoyo no hubiera sido posible llegar hasta aquí.

Roberto Gámez Pérez

Resumen

El Trabajo de Fin de Grado trata sobre el diseño e implementación de un paquete de funciones basadas en CAN bus. Dichas funciones se encargan de controlar el motor que va montado en el interior de la base que sustenta al robot ASIBOT. Este robot es desarrollado por el grupo de Robotics Lab del Departamento de Ingeniería de Sistemas y Automática de la Universidad Carlos III de Madrid.

La arquitectura de control está implementada en C/C++. Para poder comunicarnos con el driver ISCM8005 debemos utilizar una placa que tiene las funciones básicas de un PC, dicha placa es la Roboard, en la que acoplaremos una placa miniPCI que es la HICO.CAN-miniPCI, con dicha placa comunicaremos vía CAN bus con el driver que controla el motor.

El proyecto se ha desarrollado en el lenguaje de programación C/C++, la comunicación con los módulos ha sido por medio del protocolo de comunicación CANopen implementado sobre el bus de campo CAN. Este protocolo consigue una comunicación serie para el intercambio de información entre las unidades del sistema y soporta un eficiente control en tiempo real con un nivel de seguridad muy elevado. Las funciones implementadas que se encuentran desarrolladas dentro de la Roboard, son paquetes de CAN bus, que siguen el protocolo CANopen. Dichas funciones son llamadas remotamente utilizando una conexión en la que puede haber múltiples escritores para múltiples lectores (paradigma publicador/suscriptor), que es la conexión mediante YARP.

También se ha implementado un programa utilizando un lenguaje exclusivo de Technosoft, que es el TML (Technosoft Motion Language). El TML ha sido usado para desarrollar la fase de inicialización del sistema, ya que se ha programado el sistema para que la base que sustenta el ASIBOT al inicio vaya al extremo negativo y aquí sitúa el cero de referencia para el sistema de coordenadas (función homing).

Abstract

The Final Degree Project is about the design and implementation of a package of functions based on CAN bus. These functions are responsible for controlling the motor which is mounted within the base which supports the robot ASIBOT. This robot is developed by the group of Robotics Lab, Department of Systems Engineering and Automation, Universidad Carlos III de Madrid.

The control architecture is implemented in C /C++. To be able to communicate with the driver ISCM8005 we use a plate that has the basic functions of a PC, that Board is the Roboard, in which it a miniPCI board which is the HICO.CAN-miniPCI, with the plate we will communicate via CAN bus driver that controls the motor.

The project has been developed in the C programming language, the communication modules has been through the CANopen communication protocol implemented on the CAN field bus. This protocol manages a serial communication for the exchange of information between the system units and supports an efficient control in real time with a very high level of security.

The project was developed in the C/C++ programming language. Communication with the modules has been through CANopen communication protocol implemented on CAN fieldbus. This gets a serial communication protocol for exchanging information between system units and supports an efficient real-time control with a very high level of security. The implemented functions that have been developed within the RoBoard are CAN bus packages follow the CANopen protocol. These functions are called remotely using a connection that allows multiple writers for multiple readers (publisher/subscriber paradigm), using YARP.

A program that uses an exclusive language of Technosoft, which is TML (Technosoft Motion Language) has also been implemented. TML has been used to develop the system initialization phase, a homing sequence.

Capítulo 1: Introducción y objetivos

1.1 Introducción

En este primer capítulo se hará una pequeña introducción del trabajo realizado en este Trabajo de Fin de Grado. Se realizara una descripción de los aspectos más importantes acerca de la estructura que controla el sistema.

Dicho proyecto consiste en crear una aplicación que se pueda utilizar para el control de varios motores, ya sea de forma simultánea (se da una misma orden a todos los motores que se están controlando) o bien independiente (cada motor recibe una orden especifica).

Se han implementado funciones en C/C++ que se encargan de comunicar con el driver que controla el motor a través de paquetes de CAN bus, cada función tiene implementado diferentes acciones como puede ser poner el motor en modo posición o bien en modo velocidad, así como obtener los valores del encoder, mover el motor con velocidad constante, etc.

Esta aplicación desarrollada servirá para el control de un único motor que se encuentra en la base que soporta el robot ASIBOT, ya que el soporte que sirve de apoyo al robot ASIBOT y este sistema solo dispone de un motor. Dicho motor se encarga de situar el soporte del ASIBOT en diferentes puntos, según la acción que se quiera realizar.

La importancia de la aplicación, es que se puede utilizar para el control de cualquier tipo de motor, lo que simplifica mucho el trabajo. Esta aplicación se podría utilizar de base para implementar arquitecturas de control más complejas encargadas de controlar varios motores como pueden observarse en los robots asistenciales.

1.2 Objetivos

El objetivo final del proyecto es permitir el control del motor que va alojado en la base del robot asistencial ASIBOT a través de comandos simples, como puede ser escribir un número y que el brazo se vaya a un punto concreto previamente grabado.

Se deberán cumplir los siguientes objetivos especificos para este proyecto:

- Se requiere implantar un control directo que facilite el desarrollo de futuros algoritmos de control para el soporte del robot asistencial ASIBOT.
- Se requiere el desarrollo de un software que efectúa el control y monitorización de parámetros relacionados con el motor que controla el soporte del robot, mediante el empleo del protocolo de comunicaciones CANopen.
- Se desea integrar el desarrollo en una interfaz gráfica para facilitar el uso de la aplicación por parte de los usuarios.
- Se deben de realizar pruebas y ensayos para verificar el correcto funcionamiento de los elementos software y hardware.

A pesar que para esta aplicación solo se utiliza un motor, se podrían incluir mayor número de motores a controlar. Por ejemplo se podrían mandar comandos al motor de un ventilador que esté controlado en velocidad en función de la temperatura que se alcance en el sistema o en la Roboard o bien en el driver.

En este proyecto sólo están implementadas las funciones para que sean ejecutadas por el usuario, y no para que funcionen por sí mismas como un sistema inteligente que percibe el entorno y actúa en consecuencia.

1.3 Fases de desarrollo

Las fases de las que consta este proyecto de final de carrera serán tres:

1. **Fase de montaje del sistema**. Que comprende el montaje del motor y el encoder en el driver encargado del control de los motores. En esta fase también habría que montar los finales de carrera en dicho driver, así como establecer la alimentación del mismo. El conexionado de las comunicaciones también se implementa en esta fase. Así como el montaje del soporte del robot asistencial ASIBOT dentro del carril de la silla de ruedas.
2. **Fase de configuración del driver que controla los motores**. Para ello utilizaremos el EasyMotion Studio, que es un programa proporcionado por Technosoft que se utiliza para configurar el driver estableciendo la corriente de pico del motor, así como la reducción del motor. La configuración consta de tres partes:

    - La primera es la configuración de las características del motor y el encoder (Motor Setup), en la cual se escoge cual es la intensidad nominal y de pico del motor, la inductancia por fase, la relación de transmisión entre el motor y la carga, así como test de conexiones de los encoders, etc.
    - La segunda seria la configuración del driver (Driver Setup). En esta etapa se ajustan los PIDs del motor, así como el modo de control del mismo (posición, velocidad o torque). También hay que ajustar las protecciones de los motores de sobre intensidades (I^2t) y establecer como son las señales de entradas (nivel bajo o alto). Aquí podemos configurar cual quieres que sea el ancho de banda en función de lo que establezcas en las siguientes fases.
    - La tercera parte es la implementación de lo que Technosoft denomina "Motion". Que es el programa en lenguaje TML, que se utiliza para inicializar el sistema.

3. **Fase de implementación de las funciones de C/C++.** Así como el desarrollo de la aplicación que coordina todas las funciones implementadas. En esta fase hemos desarrollado los siguientes archivos:

   - Roboardbot.h: Se trata del fichero de tipo cabecera donde se encuentra declarada la *clase* de C++ que fue el desarrollo principal de este trabajo: RoboardBot. Esta clase hereda de 4 clases de funciones puras virtuales de la librería YARP_dev que agrupan funcionalidades propias del driver (gestión, control en posición, en velocidad, encoders). Las implementaciones se encuentran en ficheros que se llaman igual que las clases base con el sufijo Impl.cpp, que se explicarán a continuación.
   - IDeviceImpl.cpp: Este archivo fuente es el encargado de inicializar y cerrar el driver, así como de llamar a las diferentes funciones que se alojen en su interior. Se encarga de abrir y cerrar el sistema de control. Sería similar a una unidad de arranque, ya que se encarga de iniciar el sistema para poder ordenar posteriormente por teclado una acción.
   - IPositionImpl.cpp: Este archivo fuente es el encargado de implementar las funciones de poner el driver en modo posición, da órdenes para mover el motor en posición tanto en coordenadas absolutas, como en coordenadas relativas. También se incluyen las funciones para establecer una velocidad y aceleración de referencia.
   - IVelocityImpl.cpp: Este archivo fuente tiene como objetivo establecer en modo velocidad el driver. También se implementa la función encargada de mover el motor en modo velocidad.
   - IEncoderImpl.cpp: Este archivo fuente cuenta con las funciones relacionadas con los encoders. Aquí se encuentran las funciones encargadas de proporcionar la posición y velocidad a la que se encuentra el motor, así como resetear los encoders.
   - HelperFuncs.cpp: En este archivo fuente se implementan diferentes funciones auxiliares como la de Display, que se encarga de mostrar por pantalla los mensajes de CAN bus, o la función de read_time_out, que sirve para poder leer paquetes de CAN bus emitidos por el driver.

1.4 Medios empleados

Para el desarrollo del proyecto final de carrera he contado con los siguientes medios:

Software:

- **EasyMotion Studio**: ha sido utilizado para configurar tanto el motor, encoder y el driver que controla ambos elementos y para implementar el programa en lenguaje de programación TML.
- **EasySetup**: tiene características parecidas al EasyMotion pero más simplificado. Con este programa no se puede crear y editar movimientos, solo sirve para configurar el motor y el driver, así como ajustar los reguladores tanto de posición como de movimiento.
- **EEPROM Programmer**: se encarga de grabar el programa implementado en EasyMotion Studio en la memoria no volátil EEPROM.
- **Sistema operativo Ubuntu 11.10**: se utiliza para dar soporte a los dos programas anteriores.
- **Editor de textos "Nano":** se utiliza para implementar la arquitectura de control del motor (es decir, para implementar las funciones que conforman la aplicación.

Hardware:

- **Driver ISCM8005:** este elemento es el encargado del control del motor y del encoder. Así como dar soporte al programa en lenguaje TML. También es el encargado de ejecutar los paquetes de datos transmitidos según el protocolo CANopen implementado sobre el bus de campo CAN. Sirve para comunicar el motor y el encoder con la Roboard, ya que cuenta con dos vías de comunicación una es un puerto serie RS232 y el otro es un puerto de CAN bus. Se trata además, de un dispositivo que se ha utilizado previamente en otros proyectos de investigación del grupo Robotics Lab [1] [2].
- **CPU RoBoard RB-100:** es la encargada de soportar el sistema operativo sobre el que será ejecutado el programa implementado. Además aquí se

conectaría mediante un puerto miniPCI, la tarjeta de comunicación HICO.CAN-miniPCI.

- **Tarjeta HICO.CAN-miniPCI:** es la encargada de la comunicación según el protocolo CANopen implementado sobre el bus de campo CAN. Sin esta tarjeta sería imposible cualquier tipo de comunicación entre la placa RoBoard y el driver ISCM8005.

1.5 Estructura de la memoria

Para facilitar la lectura de la memoria, se incluye a continuación un breve resumen de cada capítulo.

En el primer capítulo se va a introducir algunos aspectos más importantes que forman el proyecto de final de carrera. Se explicaran cuales son los componentes pero sin detallarlos, este capítulo sirve para hacer una idea de la envergadura del proyecto y sus posibles aplicaciones.

En el segundo capítulo se procederá a describir con mayor nivel de detalle todos los elementos que forman el proyecto, aquí se incluyen software. Aunque nos centraremos en el siguiente software el EasyMotion Studio, también nos centraremos en aspectos como el CAN bus y el CANopen.

En el tercer capítulo se describirá los elementos de hardware utilizados como el driver ISCM8005, la tarjeta de comunicación HICO.CAN-miniPCI y la CPU Roboard.

En el cuarto capítulo se va a explicar el desarrollo de la aplicación que he implementado. Aquí vamos a centrarnos en cómo se ha combinado la trama de CAN con la programación en C/C++.

En el quinto capítulose expondrá las conclusiones del proyecto y posibles trabajos que se pueden seguir desarrollando a partir del proyecto actual.

Capítulo 2: Elementos de software

2.1 Plataformas software: software Technosoft

Para poder desarrollar el proyecto final de carrera nos hemos servido de software de la compañía Technosoft. Debido a que el driver es de dicha compañía y que no es posible configurar dicho driver sin los programas de Technosoft. Estos programas están disponibles para Windows exclusivamente, pero también se pueden utilizar programas que instalan los programas de Technosoft en sistemas operativos de Unix.

Se han utilizado principalmente dos programas, EasyMotion Studio [3] y EEPROM Programmer. Aunque también se podría haber utilizado EasySetup, pero no da la opción de crear y editar rutinas de movimiento.

El lenguaje utilizado para crear las rutinas de movimiento es un lenguaje exclusivo de Technosoft, se denomina TML (Technosoft Motion Language). Ofrece gran cantidad de configuraciones, así como gran abanico de aplicaciones a desarrollar. El único inconveniente de este lenguaje es su complejidad, ya que cuenta con un gran número de funciones y aplicaciones, para mover simplemente un motor. A pesar de estas desventajas, el propio fabricante cuenta con una serie de manuales y con un gran número de ejemplos para que puedas entender fácilmente como programar el driver.

A continuación procederemos a describir con más profundidad las herramientas de software EasyMotion Studio, EasySetup y EEPROM Programmer.

2.1.1 EasyMotion Studio y EasySetup

Con EasyMotion Studio se pude obtener el máximo beneficio de los drivers Technosoft, teniendo la capacidad de ejecutar movimientos complejos sin necesidad de un controlador de movimiento externo, gracias a su controlador de movimiento integrado.

EasyMotion Studio incluye programas como EasySetup para configurar el driver de forma rápida, también incluye un "asistente de movimiento" (Motion Wizard) para la programación de movimientos. El asistente de movimiento ofrece de forma sencilla la creación de programas escritos en TML (Lenguaje de movimientos Technosoft). Genera de forma automática las instrucciones TML, por los tanto, no es necesario saber escribir ningún código TML.

EasyMotion Studio es la herramienta recomendada para la programación de movimientos con lenguaje TML. Con el lenguaje TML, se puede simplificar realmente aplicaciones complejas, mediante la distribución de la inteligencia entre el maestro y las unidades o motores. Así, en lugar de tratar de dirigir cada paso del movimiento de un eje maestro, se puede programar las unidades o motores con TML para ejecutar tareas complejas, e informar a la unidad maestra cuando estas tareas se han completado correctamente.

EasySetup sirve para establecer de forma rápida y fácil, una configuración para una aplicación, utilizando solo dos diálogos. Esta herramienta es la recomendada cuando la programación y el control de movimientos se realizan a través de un dispositivo externo como un PC industrial o PLC, es necesario utilizar las bibliotecas de movimiento de Technosoft para PLC. La salida de EasySetup es un conjunto de datos de configuración que se pueden descargar en la memoria EEPROM del driver o conservarla en el PC. En el encendido, los datos de configuración guardados en la memoria EEPROM del driver se utilizan para las inicializaciones. También es posible recuperar la información de configuración de un driver programado con anterioridad. Este software incluye un programador de firmware para que se actualice el driver para la revisión i.

La actualización de EasyMotion Studio y de EasySetup se realiza a través de una herramienta vía Internet, con la que se puede comprobar si la versión de software está actualizada, o descargar e instalar las actualizaciones más recientes.

2.1.2 EEPROM Programmer

Todas las unidades Technosoft incluyen una memoria no volátil EEPROM. Su función es:

- Mantener los datos de configuración en un área especial llamada tabla de instalación, junto con un ID de usuario de la aplicación programable, lo que ayuda a identificar de forma rápida y segura, los datos de configuración cargados en in driver.
- Guarda los programas de movimiento en lenguaje TML y sus datos asociados, por ejemplo, las tablas de levas necesarias para aplicaciones de levas electrónicas.
- Mantener el ID (identificador) del producto de cada driver y el ID del *firmware* de la aplicación programada.

El ID de *firmware* indica que el verdadero driver debe tener el mismo número de ID *firmware* y una letra de revisión igual o superior. EEPROM Programmer [4] es una herramienta específicamente diseñada para la producción, a través de la cual usted puede:

- Programa de forma fácil y rápida la memoria EEPROM de cualquier unidad de Technosoft con todos los datos necesarios para ejecutar una aplicación especifica.
- Compruebe la integridad de los datos de la EEPROM mediante la comparación de la información leída de la memoria de driver, con la lectura de un archivo de software (.sw).
- Escribir y proteger una parte o la totalidad de la memoria EEPROM.

- Obtener información sobre el ID de configuración del driver, incluyendo la identificación del producto, la identificación del *firmware* y el ID de la aplicación.

2.2 Protocolo de comunicación: CAN

Para la realización de este Trabajo de Fin de Grado se ha empleado el protocolo de comunicación CANopen para comunicar la CPU donde se ejecutan las funciones implementadas en lenguaje de programación C/C++, con los drivers que controlan los motores. Este protocolo se basa en un bus de campo CAN, por lo tanto se procederá a explicar el funcionamiento del bus CAN [5].

2.2.1 Bus CAN

2.2.1.1 Introducción

CAN bus es un protocolo de comunicación en serie que fue desarrollado por Bosch para el intercambio de información entre unidades de control electrónicas. CAN significa Controller Area Network (Red de área de control) y Bus, se entiende como un elemento que permite transportar una gran cantidad de información.

Este sistema permite compartir una gran cantidad de información entre las unidades de control abonadas al sistema, lo que provoca una reducción importante tanto del número de sensores utilizados como de la cantidad de cables que componen la instalación eléctrica. EL objetivo principal del bus CAN es el de reducir el número de cables de comunicación y control, y ordenar la información que se traspasa entre un sistema y otro.

Fue desarrollado por Robert Bosch en 1986 y en 1993 fue estandarizado por la norma ISO 11898.

A continuación vamos a detallar las características más importantes del protocolo CAN, así como los elementos que lo componen, también se detallara el formato de dicho protocolo y su funcionamiento.

2.2.1.2 Características del protocolo CAN

En términos comunes, el protocolo CAN verifica si el bus se encuentra ocupado antes de transmitir, dando la posibilidad de que múltiples nodos puedan transmitir y recibir al mismo tiempo. Cuando se produce una colisión de mensajes en el bus, empieza lo que se conoce con el nombre de arbitraje para la recepción de los mensajes. El mensaje con mayor prioridad se recibe primero y así sucesivamente, hasta que todos los mensajes sean recibidos. El esquema de arbitraje que usa el protocolo CAN es de bit inteligente no destructivo. Esto significa que se comparan mensajes con cada bit, pero el mensaje con más prioridad no se destruye y se retransmite, solo el mensaje que no gana el arbitraje se detiene y se retransmite después. Esto minimiza el tiempo que está fuera de servicio el bus y aumenta al máximo el uso del ancho de banda disponible.

Otra ventaja importante del arbitraje del bit inteligente no destructivo, que usa el CAN. Con Ethernet, ambos transmisores se detienen cuando se detecta una colisión y una cantidad de tiempo aleatorio que debe pasar antes de que ambos prueben la retransmisión Con este elemento aleatorio de tiempo eliminado de la función del bus, es posible lograr casi el 100% de eficacia en cuanto a utilización del ancho de banda.

Con el protocolo CAN las posibilidades de transmitir y recibir un error son muy bajas. Hay dos tipos básicos de errores que son **errores de bit** y **errores de mensaje**. Un tipo de **error de bit** es un “bit de error de relleno”. El bus CAN necesita sincronizarse periódicamente, si hay cinco niveles bajos consecutivos el transmisor inserta un nivel bajo y si hay cinco niveles altos consecutivos el transmisor inserta un nivel alto. Si hay más de cinco bits consecutivos se produciría un “bit error de relleno”.

Hay tres tipos de **errores de mensaje.** El primer tipo son los “errores de verificación de suma” (“check sum”), que producen cuando la verificación cíclica de redundancia no se empareja. El segundo tipo son los “errores de formato”, que se producen cuando se detecta un bit invalido en ciertas posiciones de un mensaje CAN. Estas posiciones cruciales son bits que se utilizan dentro del CAN para detectar posibles errores a la hora de transmitir los paquetes de datos. Finalmente están los “errores de reconocimiento”, que se producen cuando un transmisor no reconoce uno de sus mensajes.

Cuando se han encontrado errores, se envía un *frame* de error. El mensaje con error se cancela de todos los nodos que lo recibieron y se actualiza el estado de error de todos los nodos, después se retransmite de nuevo el mensaje por el transmisor.

Los controladores de CAN pueden estar en tres modos posibles:

- Error activo (el modo predefinido).
- Error pasivo. Un nodo entra en este modo cuando el nodo presenta muchos errores, debido a contadores que se encargan de contabilizar el número de errores de cada nodo. Cuando se supera la asignación máxima de errores en un nodo determinado, el nodo se desactiva para evitar problemas posteriores.
- Fuera de bus. Si un nodo en particular está teniendo problemas, será aislado y así no podrá contaminar al resto de los nodos con información defectuosa.

La ISO 11898 define la capa física del CAN, es una interfaz de dos hilos en modo diferencial por un par trenzado apantallado (STP) o un par trenzado no apantallado (UTP) o un cable plano (cinta). Cada nodo usa un conector de 9-pin subD. Este protocolo permite la creación de redes con gran tolerancia de errores.

La velocidad es programable, a alta velocidad puede llegar a 1 Mbits/s, a distancias de 40 m y a baja velocidad desde 5 Kbits/s, a distancias de 10 km, por lo tanto la velocidad de transmisión de datos depende de la distancia de comunicación. Un ejemplo de transmisión usando este protocolo se muestra en la Figura 1 (se usan resistencias de terminación en cada extremo del cable).

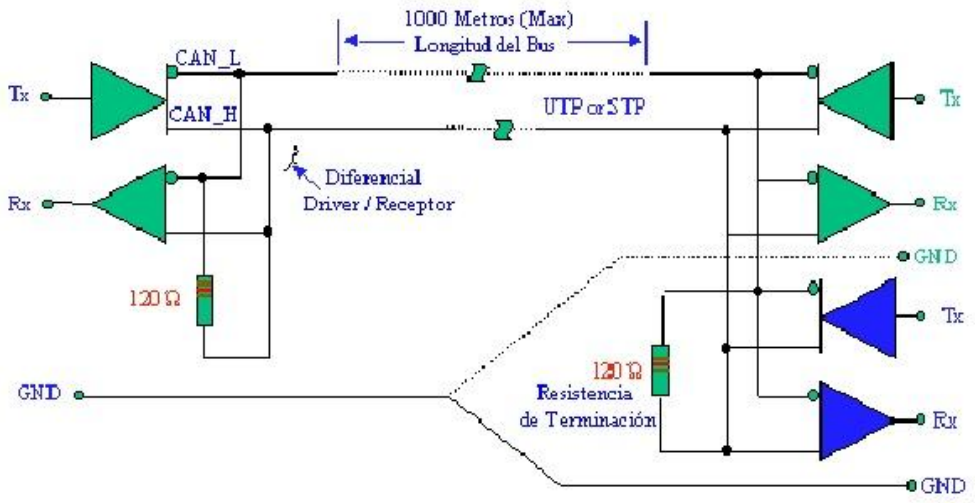


Figura 1: Modo de transmisión del protocolo CAN.

El hardware CAN cubre las dos primeras capas del modelo OSI (Open Systems Interconnection, Figura 2), que corresponden con la capa física y la capa de enlace de datos.

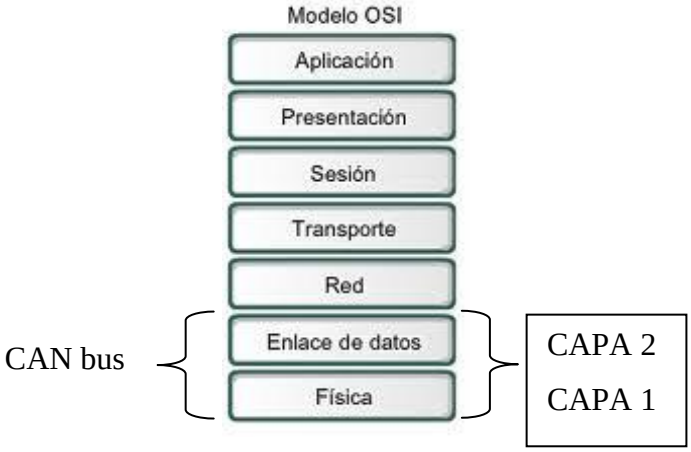


Figura 2: CAN y el modelo de referencia OSI.

La capa física es el medio físico a través del cual se transmiten los datos entre los nodos del sistema. La única restricción de esta capa es que los nodos deben emplear el mismo medio físico (par trenzado, fibra óptica, etc.).

La capa de enlace de datos se encarga principalmente del protocolo de comunicaciones. Controla la estructura de los mensajes, chequea y señaliza errores,

etc. Dentro de dicha capa se decide si el bus esta libre para ser utilizado o si se ha iniciado una transmisión. Para cubrir las restantes capas del modelo OSI vamos a recurrir al protocolo CANopen del que hablaremos posteriormente.

2.2.1.3 Componentes del sistema CAN

El sistema CAN tiene los siguientes elementos:

- **Medio físico (cable):** el medio más utilizado es el par trenzado (ver Figura 3), aunque se puede recurrir a otros como la fibra óptica. La información circula por dos cables trenzados que unen todas las unidades de control que forman el sistema. Esta información se transmite por diferencia de tensión entre los dos cables, de forma que un valor alto de tensión representa un 1 y un valor bajo de tensión representa un 0. La combinación de unos y ceros conforman el mensaje a transmitir. En uno de los cables los valores de tensión oscilan entre 0 V y 2,25 V, por lo que se denomina cable L (low) y por el otro cable los valores de tensión oscilan entre 2,75 V y 5 V, por lo que se denomina cable H (high). Si se interrumpe la línea H o se deriva a masa, el sistema trabajara con la señal Low con respecto a masa, si se interrumpe o deriva la línea L, ocurriría lo contrario. Gracias a esto el sistema podría funcionar con uno de los cables cortados o comunicado a masa, solo se quedaría fuera de servicio en el caso de que los dos cables fueran cortados. Es importante tener en cuenta que el trenzado entre las líneas sirve para anular los campos magnéticos, por esto los cables no se pueden modificar su longitud, ni su paso.

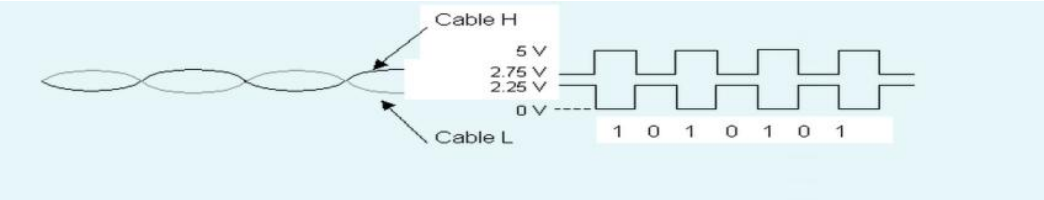


Figura 3: Cables trenzados.

- **Elemento de cierre o terminador**: son resistencias conectadas a los extremos de los cables trenzados (cables H y L). Su valor se calcula de forma empírica y permite adecuar el funcionamiento del sistema en función a la longitud del cable y del número de unidades de control que componen el sistema, ya que estas resistencias evitan el fenómeno de reflexión, que puede perturbar el mensaje. Estas resistencias se pueden alojar en el interior de las unidades de control (ver Figura 4).

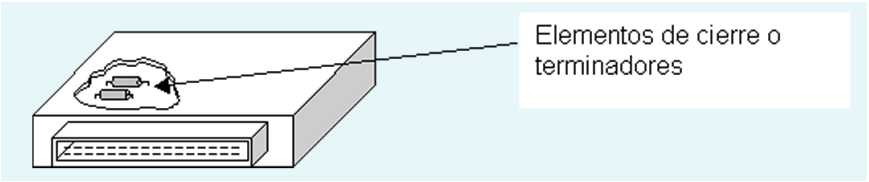


Figura 4: Elemento de cierre o terminador.

- **Controlador**: es el elemento encargado de la comunicación entre el microprocesador de la unidad de control y el transmisor-receptor. Acondiciona la información que entra y sale entre ambos componentes. Este elemento se sitúa en la unidad de control, por lo que existen tantos como unidades de control hay conectadas en el sistema. El controlador trabaja con niveles de tensión muy bajos y determina la velocidad de transmisión de los mensajes. También interviene en la sincronización entre las diferentes unidades de mando para la correcta emisión y recepción de los mensajes.

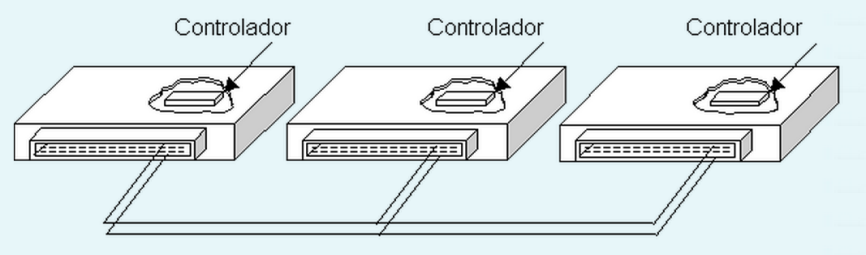


Figura 5: Controlador.

- **Transmisor / Receptor**: es el elemento que se encarga de recibir y transmitir los datos, además de acondicionar y preparar la información para que se pueda utilizar por los controladores. Esta preparación consiste en situar los

niveles de tensión de forma correcta, amplificando la señal cuando la información es volcada en la línea, reduciéndola cuando es recogida de la misma y suministrada al controlador. Este elemento es básicamente un circuito integrado que se sitúa en cada unidad de control que compone el sistema, trabaja con intensidades de 0,5 A y no modifica el contenido del mensaje. Funcionalmente está situado entre los cables que forman la línea CAN bus y el controlador. (Ver Figura 6).

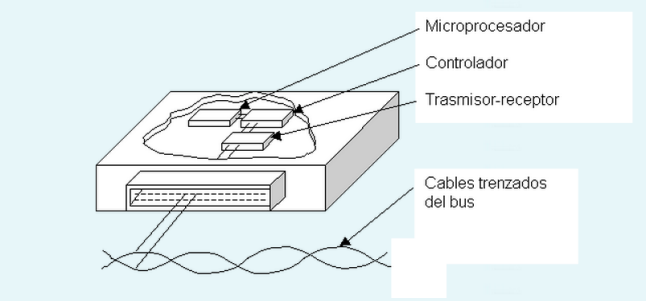


Figura 6: Transmisor / Receptor.

2.2.1.4 Estructuras de las tramas CAN

El sistema de CAN bus está orientado hacia el mensaje y no al destinatario. La transmisión de la información en la línea se hace en forma de mensaje estructurado en la cual una parte del mensaje es un identificador que indica el tipo de dato que contiene.

El mensaje es una sucesión de ‘0’ y ‘1’, que se representan mediante diferentes niveles de tensión en los cables del CAN bus y que se denominan “bit”. El mensaje posee una serie de campos de diferente tamaño (numero de bits) que permiten llevar a cabo el proceso de comunicación entre las unidades de mando según el protocolo definido por Bosch para el CAN bus, que facilitan desde identificar la unidad de mando, como indicar el principio y el final del mensaje, mostrar datos, etc.

Los mensajes son introducidos en la línea con una cadencia que oscila entre los 7 y 20 ms dependiendo de la velocidad del área y de la unidad de mando que los introduce.

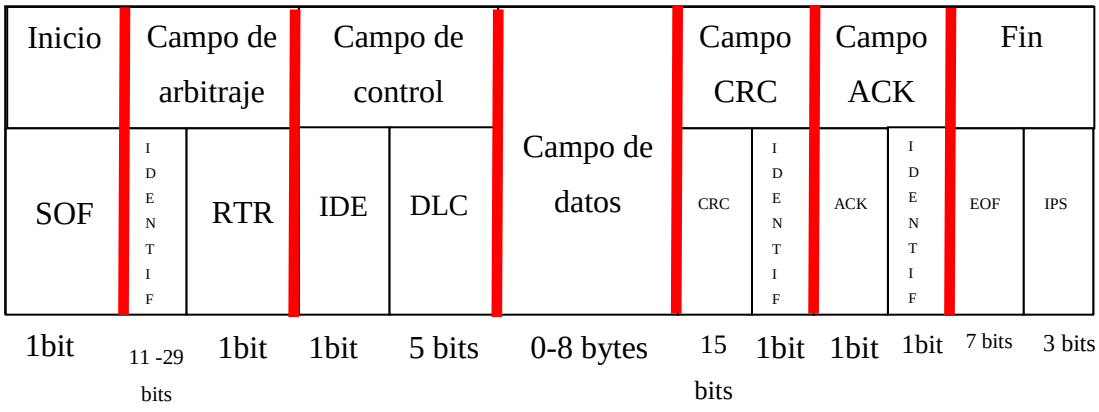


Figura 7: Formato del mensaje CAN.

- **Campo de inicio del mensaje (SOF):** el mensaje se inicia con un bit dominante, cuyo flanco descendente es utilizado por las unidades de mando para sincronizarse entre sí.
- **Campo de arbitraje:** los 11 bits primeros de este campo se emplean como identificador que permite reconocer a las unidades de mando la prioridad del mensaje. Cuanto más bajo sea el valor del identificador más alta es la prioridad, y por lo tanto determina el orden en que se introducen los mensajes en la línea. El bit RTR indica si el mensaje contiene datos (RTR =0) o si se trata de una trama remota sin datos (RTR=1). Una trama con datos siempre tiene más prioridad que una trama remota. La trama remota se usa para pedir datos a otras unidades de mando o bien porque se necesitan o para realizar un chequeo.
- **Campo de control:** este campo informa sobre las características del campo de datos. El IDE indica cuando es un ‘0’ que se trata de una trama estándar y cuando es un ‘1’ que es una trama extendida. Los cuatro bit que componen el campo DLC indican el numero de bytes contenido en el campo de datos. La diferencia entre una trama estándar y una trama extendida es que la primera tiene 11 bits y la segunda 29 bits. Ambas tramas pueden coexistir eventualmente, y esto se debe a que existen dos versiones de CAN.

- **Campo de datos:** en este campo aparece la información del mensaje con los datos que la unidad de mando correspondiente introduce en la línea CAN bus. Puede contener entre 0 y 8 bytes (de 0 a 64 bits).
- **Campo de aseguramiento (CRC):** este campo tiene una longitud de 16 bits y es utilizado para la detección de errores por los 15 primeros, mientras el último siempre es un bit recesivo (1) que delimita el campo CRC.
- **Campo de confirmación (ACK):** el campo ACK está compuesto por dos bits que son siempre transmitidos como recesivos (1). Todas las unidades de mando que reciben el mismo CRC modifican el primer bit del campo ACK por uno dominante (‘0’), de forma que la unidad de mando que está todavía transmitiendo reconoce que al menos alguna unidad de mando ha recibido un mensaje escrito correctamente. De no ser así, la unidad de mando transmisora interpreta que su mensaje presenta un error.
- **Campo de final de mensaje (EOF):** este campo indica el final del mensaje con una cadena de 7 bits recesivos.

En determinado mensajes se pueden producir largas cadenas de ceros o unos, y que esto provoque una pérdida de sincronización entre unidades de mando. El protocolo CAN resuelve esta situación insertando un bit de diferente polaridad cada cinco bits iguales, cada cinco ‘0’ se inserta un ‘1’ y viceversa. La unidad de mando que utiliza el mensaje, descarta un bit posterior a cinco bits iguales. Estos bits reciben el nombre de bit *stuffing*.

2.2.1.5 Funcionamiento del protocolo CAN

Las unidades de mando que se conectan al sistema CAN bus son las que necesitan compartir información, pertenezcan o no a un mismo sistema.

Todas las unidades de control reciben y filtran el mensaje, pero solo lo emplean las unidades de control que necesitan dicho dato. Todas las unidades de control que forman parte del sistema tienen la capacidad tanto de introducir como de recoger datos de la línea.

Cuando el bus queda libre cualquier unidad de control puede empezar a transmitir un nuevo mensaje. Si se da el caso de que una o varias unidades quieran introducir mensajes al mismo tiempo, lo hará la que tenga mayor prioridad, que viene indicado por el identificador.

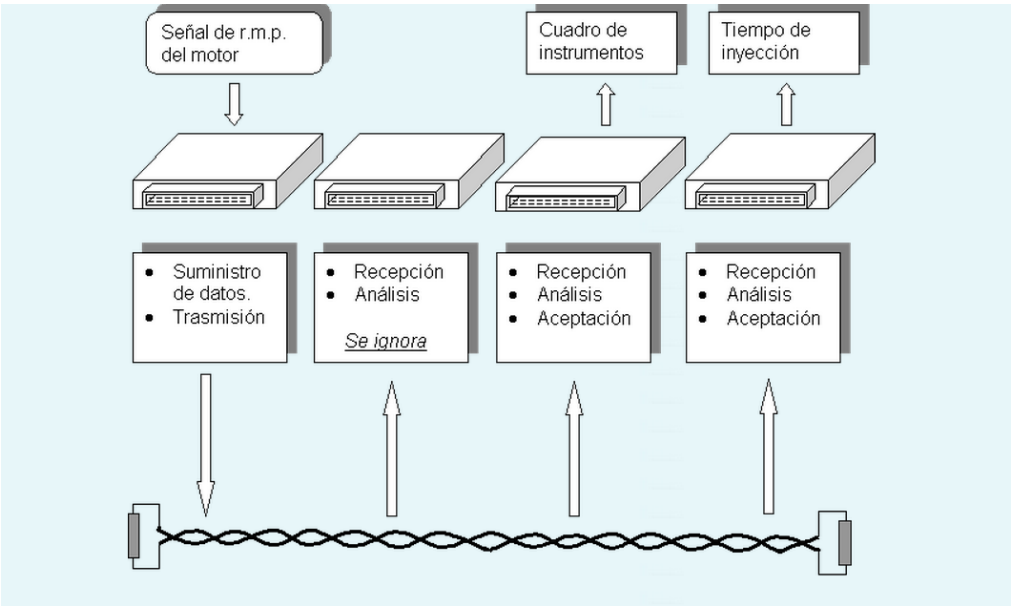


Figura 8: Ejemplo de funcionamiento del protocolo CAN.

El proceso de transmisión de datos evoluciona siguiendo un ciclo que se compone de varias fases (ver Figura 8):

- **Suministro de datos:** una unidad de mando recibe la información proporcionada por los sensores que tiene asociados (temperatura del motor, velocidad, puerta cerrada, etc.) y su microprocesador pasa la información al controlador donde es gestionada y acondicionada para ser mandada al transmisor-receptor que será donde se transforma en señales eléctricas.
- **Transmisión de datos:** el controlador de dicha unidad transfiere los datos y su identificador junto con la petición de inicio de transmisión, aceptando la responsabilidad de que el mensaje sea transmitido de forma correcta a todas las unidades de mando asociadas. Para transmitir el mensaje, el bus se debe encontrar libre, y en el caso que se produzca una colisión con otra unidad de mando que intenta transmitir simultáneamente, tener mayor prioridad. Una vez ocurrido esto, el resto de unidades de control se convierten en receptoras.

- **Recepción del mensaje:** Cuando todas las unidades de mando reciben el mensaje, verifican el identificador para determinar si el mensaje va a ser utilizado por ellas. Las unidades de control que necesitan dichos datos, procesan el mensaje, si no lo necesitan, el mensaje es ignorado.

Hay sistemas en los cuales hay datos que deben de ser transmitidos con mayor urgencia y frecuencia que otros. La prioridad de un mensaje sobre otro viene determinada en el identificador de cada mensaje, expresado en binario. El identificador con menor valor en binario tendrá mayor prioridad.

El acceso al bus se resuelve mediante un sistema de arbitraje inteligente, teniendo el nivel dominante (‘0’) prioridad sobre el recesivo (‘1’). El nodo que intenta escribir un bit recesivo pierde en este arbitraje frente a un nodo que intente escribir un bit dominante, quedando como receptor y no pudiendo acceder al bus hasta que no se encuentre nuevamente disponible. (Ver Figura 9).

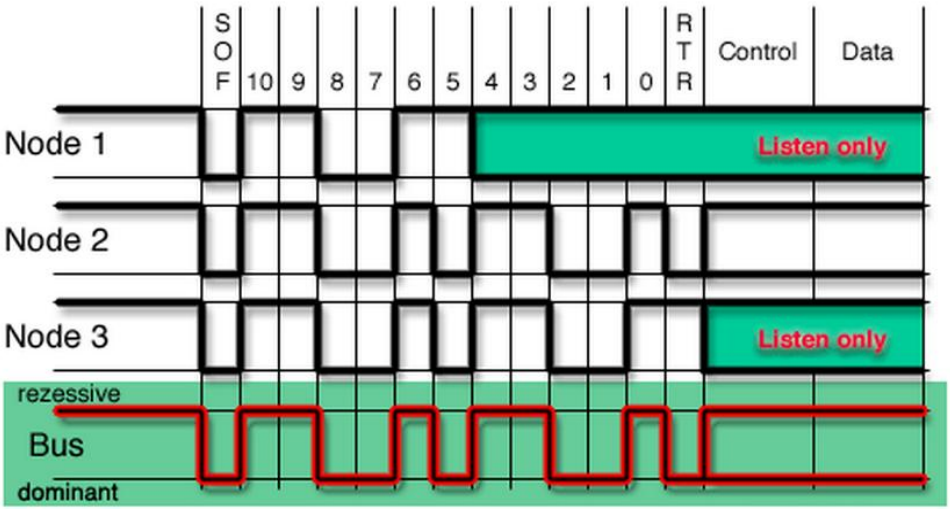


Figura 9: Ejemplo de arbitraje de acceso al bus.

2.2.1.6 Diagnosticar el bus CAN

Los sistemas de seguridad que incorpora el bus CAN hacen que las probabilidades de fallo en el proceso de comunicación sean bajas, pero cabe la posibilidad que las unidades de mando, cables y contactos presenten fallos. Se debe tener en cuenta que una unidad de mando que se encuentre abonada al bus CAN y que de fallos no impide que el sistema trabaje con normalidad. No será posible llevar a cabo las funciones que implican el uso de información proporcionada por la unidad averiada, pero si del resto de unidades.

Es posible localizar fallos en el bus CAN utilizando sistemas de auto diagnosis, donde se podrá averiguar desde el estado de funcionamiento del sistema hasta las unidades asociadas al mismo.

Tambien se puede emplear como herramienta de diagnóstico programas informáticos como CANking, CANAlyzer, etc. Estos programas visualizan el tráfico de datos en el bus e indican el contenido de los mensajes.

Aunque la herramienta más adecuada y asequible sea el osciloscopio digital con dos canales, memoria y un ancho de banda de 20 MHz. Esta herramienta permite visualizar perfectamente los mensajes utilizando base de tiempos de 100 µs y una de base de tensión de 5 V. Si se emplea un osciloscopio se deben eliminar los bits stuff (el que se añade después de cinco bits iguales).

2.2.1.7 Detección de errores del protocolo CAN

Los errores que se pueden dar en el protocolo CAN son los siguientes:

- **Error de bit:** el error de bit se produce cuando el valor del bit recibido es diferente que el valor del bit de transmisor.

- **Error de relleno:** el error de relleno es detectado por un recibidor si seis valores de bits consecutivos son recibidos durante un campo de mensajes que deben ser codificados por el bit de relleno.
- **Error de CRC:** un erro de CRC es detectado por el recibidor si el calculador CRC es diferente del recibidor CRC en el campo de secuencia del CRC.
- **Error de forma:** el error de forma es detectado por el recibidor si el bit de campo de la forma arreglada contiene uno o más bits ilegales.
- **Error de reconocimiento:** un error de reconocimiento es detectado por el transmisor si no recibe un bit dominante durante el *slot* ACK.

2.2.2 Protocolo CANopen

2.2.2.1 Introducción

Como se ha visto en el apartado anterior el bus de campo CAN solo define la capa física y la capa de enlace de datos del modelo OSI, por lo tanto sería necesario definir como se asignan y utilizan los identificadores y datos de los mensajes transmitidos. El protocolo CANopen está basado en CAN e implementa la capa de aplicación (capa 7). (Ver Figura 10).

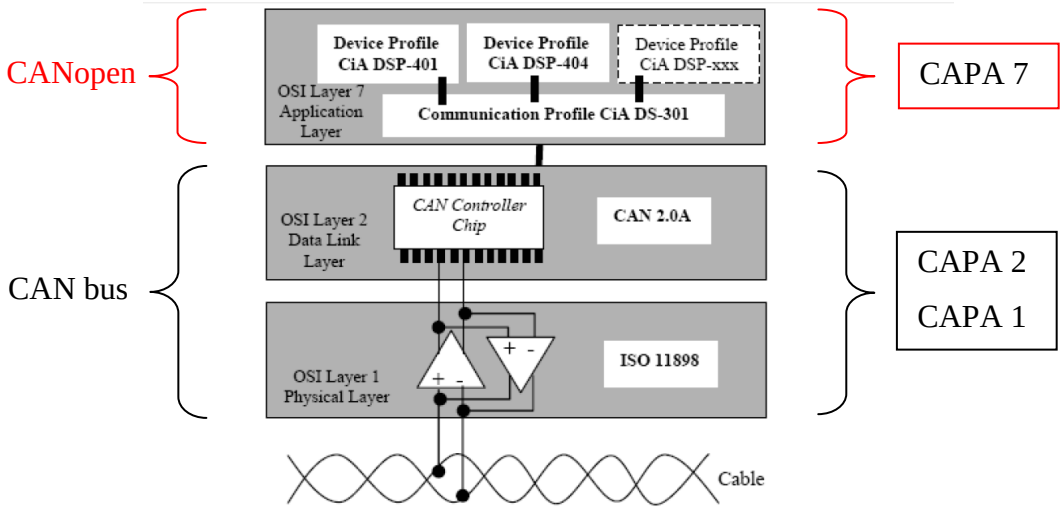


Figura 10: CANopen y el modelo de referencia OSI.

A continuación se describirá la capa de aplicación CAL para las redes CAN, en la que se basa el protocolo CANopen. Después se explicará dicho protocolo y los perfiles que define.

2.2.2.2 CAL (CAN Application Layer)

Fue de las primeras especificaciones producidas por CiA (CAN in Automation), basada en un protocolo existente desarrollado originalmente por Philips Medical Systems. Ofrece un ambiente orientado a objetos para el desarrollo de aplicaciones distribuidas de cualquier tipo, basadas en CAN.

Esta capa está compuesta por los siguientes cuatro servicios:

- **CMS (*CAN-based Message Specification*):** ofrece objetos de tipo variable, evento y dominio para diseñar y especificar como se accede a la funcionalidad de un dispositivo través de su interfaz CAN.
- **NMT (*Network ManagemenT*):** proporciona servicios para la gestión de la red. Realiza tareas de inicializar, arrancar, parar o detectar fallos en los nodos. Se basa en el concepto de maestro-esclavo, habiendo sólo un NMT maestro en la red.
- **DBT (*DistriBuTor*):** se encarga de asignar de forma dinámica los identificadores CAN (de 11 bits), también llamados COB-ID (Communication Object Identifier). También se basa en el modelo maestro-esclavo, existiendo sólo un DBT maestro.
- **LMT (Layer Management):** permite cambiar ciertos parámetros de las capas como por ejemplo el identificador de un nodo (Node-ID) o la velocidad del bus CAN.

CMS define 8 niveles de prioridad en sus mensajes, cada uno con 220 COB-IDs, ocupando desde el 1 al 1760. Los identificadores restantes (0, 1761-2031) se reservan para NMT, DBT y LMT, como se muestra en la Tabla 1. En una red CAN el mensaje con mayor prioridad es es el que posee el COB-ID mas pequeño.

COB-ID	Utilización	Cantidad
0	NMT Start/Stop	1
1-220	Objetos CMS Prioridad 0	220
221-440	Objetos CMS Prioridad 1	220
441-660	Objetos CMS Prioridad 2	220
661-880	Objetos CMS Prioridad 3	220
881-1100	Objetos CMS Prioridad 4	220
1101-1320	Objetos CMS Prioridad 5	220
1321-1540	Objetos CMS Prioridad 6	220
1541-1760	Objetos CMS Prioridad 7	220
1761-2015	Monitoreo dispositivos NMT	255
2016-2031	Servicios de NMT, LMT y DBT	16

Tabla 1: Distribución de los COB-IDs en CAL.

CAL proporciona todos los servicios de gestión de red y mensajes del protocolo pero no define los contenidos de los objetos CMS ni los tipos de objetos. Aquí es donde entra en juego el CANopen.

CANopen está por encima de CAL y utiliza un subconjunto de sus servicios y protocolos de comunicación. Proporcionando el control distribuido de un sistema utilizando los servicios y protocolos de CAL. Siendo el Diccionario de Objetos (OD, Device Object Dictionary) la pieza clave de CANopen.

2.2.2.3 Modelo del nodo CANopen

La siguiente figura muestra la estructura de un nodo de CANopen (ver Figura 11).

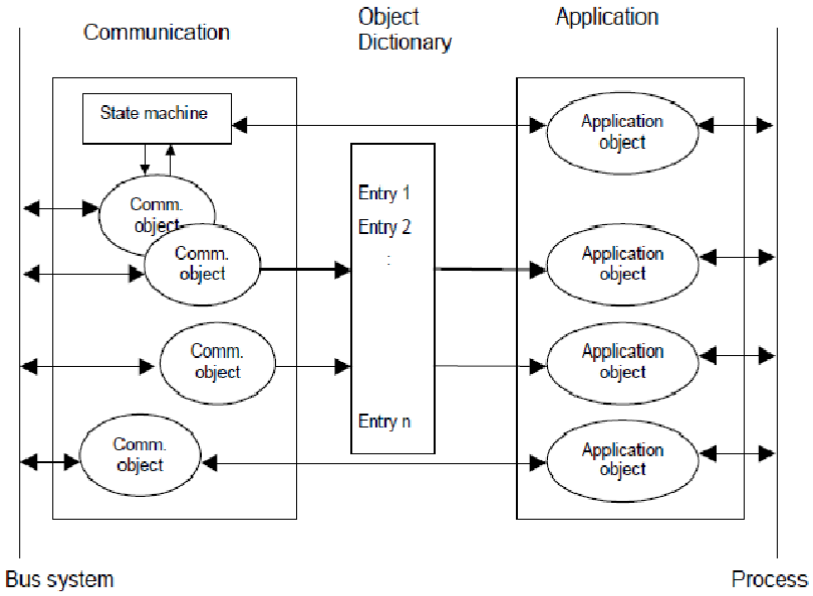


Figura 11: Modelo del nodo CANopen.

- **Comunicación:** esta unidad está encargada de suministrar los objetos de comunicación y de la correcta funcionalidad para transportar los campos de datos por el bus.
- **Diccionario de objetos (Object Dictionary):** es una colección de todos los campos de datos que influyen en el desarrollo de aplicaciones de objetos, los objetos de comunicación y la máquina de estados usada en el nodo. Es decir, es un grupo ordenado de objetos que describe de forma estandarizada la funcionalidad de cada nodo.
- **Aplicación:** se encarga de controlar la funcionalidad del nodo con respecto a la interacción con el entorno de procesos.

El Diccionario de Objetos cumple con la función de interfaz entre la comunicación y la aplicación. La descripción completa de las aplicaciones de un dispositivo con respecto a las entradas en el diccionario se conoce como perfil del dispositivo.

2.2.2.4 Modelo de comunicación

En el modelo de comunicación se definen los objetos de comunicación, servicios y modos de transmisión de mensajes. Se pueden soportar con este modelo tanto transmisiones síncronas como asíncronas.

A través de la transmisión síncrona, la adquisición de datos se puede realizar de forma totalmente coordinada. Los mensajes síncronos son transmitidos con respecto a un mensaje de sincronización predefinido, mientras que los mensajes asíncronos se pueden transmitir en cualquier momento.

Es posible definir tiempos de inhibición para la comunicación, que consisten en establecer el tiempo mínimo que tiene que transcurrir entre dos servicios consecutivos del mismo objeto de datos. Esto garantiza que los mensajes de baja prioridad tengan acceso a la red durante el tiempo de inhibición.

Existen tres tipos de comunicación:

- **Comunicación Master/Slave:** solo puede haber un maestro en la red. El resto de nodos del bus se consideran esclavos. En este tipo de comunicación se definen dos modelos (con confirmación y sin confirmación) en los que el maestro emite una petición y los esclavos responden a esta petición.

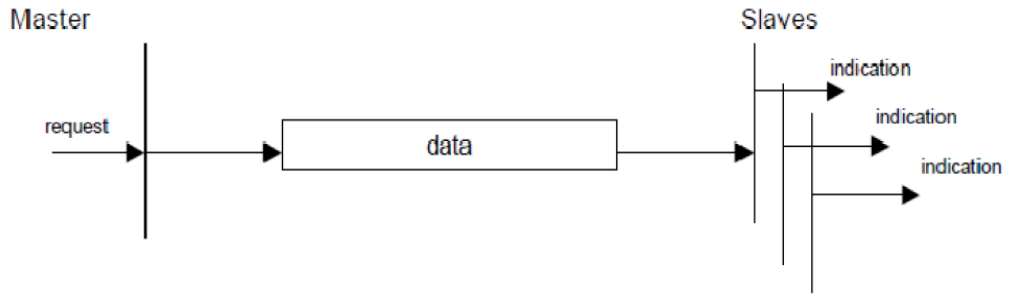


Figura 12: Comunicación Master/Slave sin confirmación.

En el modelo sin confirmación, el maestro emite un mensaje que será recibido por todos los nodos esclavos conectados al bus y solo será procesado por los dispositivos cuyos filtros estén configurados para reconocer el identificador del mensaje (Ver Figura 12). El esclavo no emitirá una nueva trama debido a la recepción del mensaje emitido por el maestro.

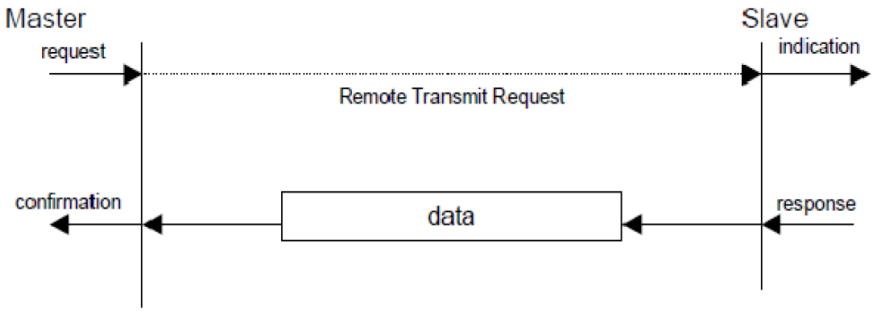


Figura 13: Comunicación Master/Slave con confirmación.

El modelo confirmado consiste en una petición de datos por parte del maestro al esclavo, por lo que si hay confirmación (ver Figura 13). La trama emitida por el maestro es del tipo RTR y por tanto no contiene datos. Este modo de comunicación no tiene carácter mandatorio.

- **Comunicación Client/Server:** este tipo de comunicación se da entre dos nodos, donde uno es el cliente y el otro el servidor (ver Figura 14). El cliente emite una petición de carga o descarga de datos pidiéndole al servidor que realice una determinada tarea. Después de llevar a cabo la tarea el servidor emite una respuesta a la petición.

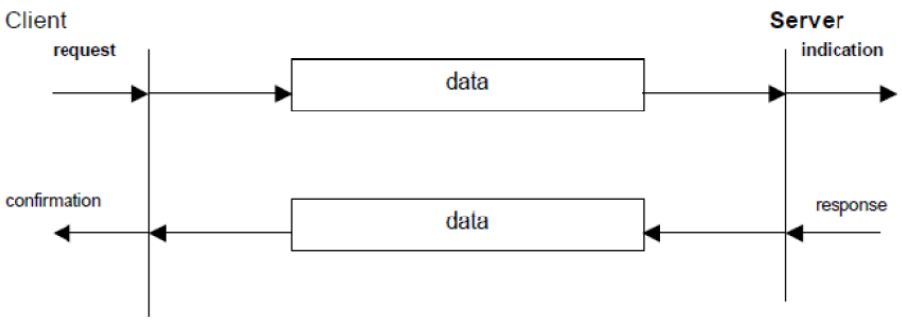


Figura 14: Comunicación Client/Server.

En este modelo se establece una comunicación punto a punto en el que solo intervienen dos de los nodos conectados al bus.

- **Comunicación Producer/Consumer:** en este tipo de comunicación un nodo actúa de productor del mensaje y cero, uno o más nodos como consumidores del mensaje. Dentro de esta comunicación se pueden distinguir dos tipos que son el "*Push model*" (es un servicio sin confirmar) y el "*Pull model*" (es un servicio confirmado).

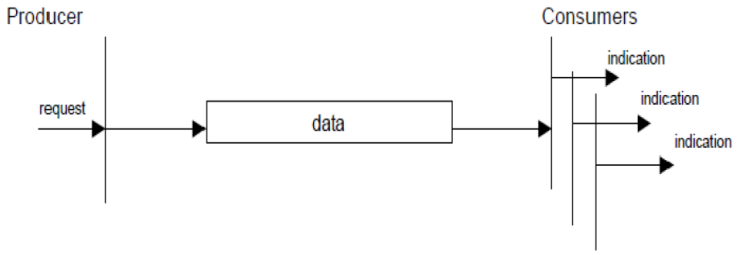


Figura 15: Push model.

El modelo Push es utilizado por los dispositivos productores para transferir datos en tiempo real por el bus (ver Figura 15). Es similar al modelo maestro/esclavo, pero el modelo maestro/esclavo se utiliza para controlar el estado de los nodos de la red.

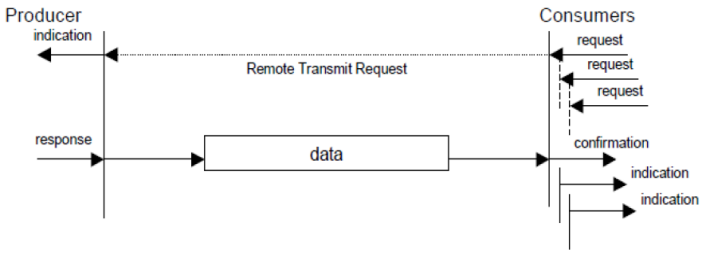


Figura 16: Pull model.

El modelo Pull se utiliza para solicitar datos a un nodo productor (ver Figura 16). Este modelo es parecido al modelo maestro/esclavo confirmado, la principal diferencia entre estos modelos es en los tipos de datos que solicitan, ya que el modelo Pull es utilizado para obtener datos en tiempo real de control, mientras que el modelo maestro/esclavo confirmado es utiliza para confirmar el estado del esclavo.

2.2.2.5 Objetos de comunicación

Los objetos de comunicación quedan descritos por los servicios y protocolos. El tráfico de datos por el bus relacionado con un dispositivo CANopen quedaría dividido en dos clases:

- **Service Data Object (SDO):** incluye todo lo relacionado con los parámetros de configuración. Son mensajes de servicio de baja prioridad y son utilizados para leer y escribir cualquiera de las entradas del diccionario de objetos de un dispositivo CANopen.
- **Process Data Object (PDO):** abarca todo lo relacionado con las operaciones en tiempo real del dispositivo. Son mensajes de alta prioridad y son utilizados para el intercambio de datos de proceso en tiempo real.

A continuación se pueden observar en la Tabla 2 las principales diferencias entre PDO y SDO.

PDO (Process Data Object)	SDO (Service Data Object)
Usado para el intercambio de datos en tiempo real.	Usado para mensajes de configuración.
Posibilidad de transmisión cíclica o a cíclica.	Transmisión asíncrona normalmente.
Optimizado para alta velocidad.	Baja velocidad.
Mensajes de prioridad alta.	Mensaje de prioridad baja.
Solo 8 bytes por mensaje.	Posibilidad de multi-telegramas.
Campos de datos.	Campos de datos.
Estructura configurable mediante el canal de servicios.	Estructura fija.
Rápido	Lento.

Tabla 2: Diferencias entre PDO y SDO.

Adicionalmente a los mensajes PDO y SDO, existen otros tipos de objetos:

- **Synchronisation Object (SYNC):** es utilizado para sincronizar todas las aplicaciones del bus.
- **Time Stamp Object (TIME):** provee una referencia de tiempo común a todos los nodos del bus.
- **Emergency Object (EMCY):** envía información de errores internos en los nodos si estos se producen y el objeto esta implementado.
- **Network Management Object (NMT):** se encarga de controlar y enviar información de la máquina de estados de los nodos que estén conectados al bus.

2.2.2.5.1 Process Data Object (PDO)

Los *Process Data Object* (PDO) se usan para transferir datos en **tiempo real** entre el master CANopen y las unidades. También pueden ser enviados desde un nodo productor a uno o más consumidores.

Los PDOs son servicios no confirmados que se desarrollan sin sobrecarga del protocolo. Aunque los PDOs no están indexados se corresponden con entradas en el Diccionario de Objetos y provee de una interfaz a los objetos de la aplicación. El tipo de dato y mapeado de un PDO para cada objeto de aplicación está determinada por defecto dentro de la estructura mapeada del Diccionario de Objetos de cada dispositivo.

Si el nodo soporta el mapeado de PDO variable, el formato y contenido de los mensajes PDO puede ser configurado entre el servidor y cliente en la fase de inicialización. Esta operación se realiza mediante la utilización de mensaje SDO correspondientes a cada entrada en el Diccionario de Objetos que se desee modificar. Existen dos tipos de PDOs. El primero se usa para transmitir un PDO (TPDO) y el segundo se usa para recibir un PDO (RPDO).

Los dispositivos que soportan la transmisión de mensajes PDOs se consideran como productores de PDO, mientras que los que reciben el mensaje PDO se consideran consumidores de PDO. Los PDOs quedan descritos por medio de los parámetros de comunicación PDO y por medio de los parámetros de mapeado PDO.

Los parámetros de comunicación describen la capacidad de comunicación de los PDOs (COB-ID utilizado por el PDO, el tipo de transmisión, tiempos de inhibición y temporización). Los parámetros de mapeado contienen información acerca del contenido de los PDOs, contiene una lista de objetos del Diccionario de objetos contenidos en el PDO, incluyendo su tamaño en bits.

Los índices de mapeado de las correspondientes entradas en el Diccionario de Objetos, son los mostrados en la Tabla 3.

	TPDO1	TPDO2	TPDO3	TPDO4	RPDO1	RPDO2	RPDO3	RPDO4
Mapeo	1A00h	1A01h	1A02h	1A03h	1600h	1601h	1602h	1603h
Comunicación	1800h	1801h	1802h	1803h	1400h	1401h	1402h	1403h

Tabla 3: Objetos de diccionario para configurar los PDOs.

Para la transmisión de mensajes PDOs se pueden diferenciar dos modos de transmisión, la transmisión síncrona y la transmisión asíncrona. Para llevar a cabo la sincronización de todos los dispositivos de la red CANopen se transmiten objetos de sincronismo (SYNC) de forma periódica. Los mensajes síncronos son enviados dentro de una ventana de tiempo predefinida después de cada objeto de sincronismo. En la siguiente figura se pueden observar los dos modos de transmisión. (Ver Figura 17).

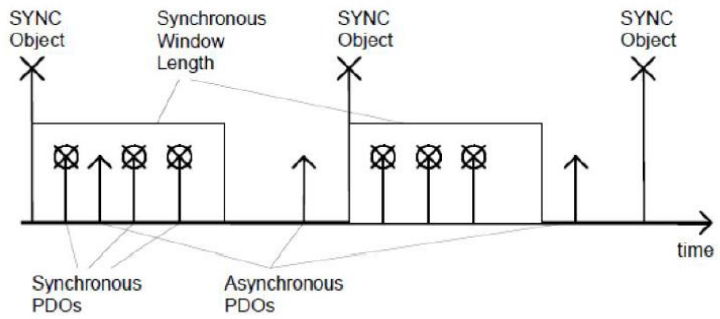


Figura 17: Transmisión de PDOs síncronos y asíncronos.

El modo de transmisión y el modo de accionamiento (evento que provoca la transmisión de un PDO) se especifican en los parámetros de tipo de transmisión de un PDO.

Para los TPDOs se especifica la tasa de transmisión en forma de un factor en base al periodo de transmisión del objeto de sincronismo. Un factor '0' el mensaje se transmite de forma aciclica después del objeto de sincronismo, es decir, solo se transmite si se produce un evento antes del objeto de sincronismo. Un factor '1' el mensaje será transmitido después de cada objeto de sincronismo. Si el factor es 'n' el mensaje se transmitirá después de 'n' objetos de sincronismo. En el caso de que el factor sea '254' o '255' el modo de transmisión será asíncrono.

A continuación vamos a describir los pasos a seguir para la realización del mapeado dinámico de un PDO, esto sirve para cambiar el mapeado por defecto de un PDO.

1. **Deshabilitar el PDO.** En los objetos de mapeado (índices 1600h-1603h para RPDOs y 1A00h-1A03h para TPDOs) se pone el primer subíndice a '0'. Esto deshabilitara el PDO.
2. **Cambiar los parámetros de comunicación.** Si es necesario cambia los parámetros de comunicación del PDO (índices 1400h-1403h para RPDOs y 1800h-1803h para TPDOs): el COB-ID, el tipo de transmisión o el acontecimiento que provoca la transmisión.
3. **Mapear los nuevos objetos.** Escribe en los objetos de mapeado de los PDOs (índices 1600h-1603h para RPDOs y 1A00h-1A03h para TPDOs) en los subíndices del '1' al '8' la descripción de los objetos que serán mapeados. Se pueden mapear hasta ocho objetos de un byte de tamaño.
4. **Habilitar el PDO.** En el subíndice '0' de los PDOs asociados al objeto de mapeado (índices 1600h-1603h para RPDOs y 1A00h-1A03h para TPDOs) escribir el numero de objetos mapeados.

Ahora se mostrará un ejemplo más visual para mapear un TPDO3 de numero de eje '10' con ***Control Word*** (índice 6040h) y ***Modes of Operation*** (índice 6060h).

1. **Deshabilitar el PDO.** Escribir ‘0’ en el objeto de índice 1602h, subíndice ‘0’, esto deshabilitara el PDO.

COB-ID	60A
Data	2F 02 16 00 00 00 00 00

Tabla 4: SDO accede al objeto 1602h, subíndice 0, 8-bit valor 0.

2. **Cambia los parámetros de comunicación.** No sería necesario.
3. **Mapear los nuevos objetos.**
   a. Escribir en el objeto de índice 1602h, subíndice ‘1’ la descripción de ***Control Word***:

Índice	Subíndice	Extensión	
6040h	00h	10h	60400010h

Tabla 5: Control Word índice 6040h, subíndice ‘0’.

COB-ID	60A
Data	23 02 16 01 10 00 40 60

Tabla 6: SDO accede al objeto 1602h, subíndice 1, 32-bit valor 60400010h.

   b. Escribir en el objeto de índice 1602h, subíndice ‘2’ la descripción de ***Mode of Operation***:

Índice	Subíndice	Extensión	
6060h	00h	08h	60600008h

Tabla 7: Mode of Operation índice 6060h, subíndice ‘0’.

COB-ID	60A
Data	23 02 16 02 08 00 60 60

Tabla 8: SDO accede al objeto 1602h, subíndice 2, 32-bit valor 60600008h.

4. **Habilitar el PDO.** Ajusta el objeto 1602h, subíndice ‘0’ con el valor ‘2’.

COB-ID	60A
Data	2F 02 16 00 02 00 00 00

Tabla 9: SDO accede al objeto 1602h, subíndice ‘0’, 8-bit valor ‘2’.

2.2.2.5.2 Service Data Object

Los *Service Data Object* (SDO) son mensajes de baja prioridad. Se utilizan para leer y escribir cualquiera de las entradas del Diccionario de Objetos de un dispositivo CANopen. Debido a su baja prioridad se suelen utilizar para realizar configuraciones previas en los dispositivos de la red en lugar de ser utilizados para la transferencia de datos de proceso.

Los *Service Data Object* permiten comunicar los objetos de datos entre el maestro y los demás nodos por medio del acceso al Diccionario de Objetos del dispositivo CANopen siguiendo el modelo de comunicación *Client/Server*, donde el nodo propietario ejerce la función de servidor.

Como los tipos de datos y tamaños de los mismos pueden variar desde un simple booleano hasta archivos de datos más complejos, el SDO se utiliza para la transmisión de grupos de datos múltiples desde el cliente al servidor o viceversa estableciendo un canal de comunicación **punto a punto**.

Para todos los modos de transmisión de SDOs el cliente es quien inicia la comunicación, pero si se produce un error tanto el cliente como el servidor pueden llevar a cabo la interrupción de la transmisión del SDO.

Aunque los dispositivos CANopen pueden soportar la implementación de más de un SDO, por defecto sólo se implementa un SDO en cada servidor.

El SDO está definido por los parámetros de comunicación (SDO *Comunication Parameter*) que describen la capacidad de comunicación entre los *Server*-SDOs (SSDO) y *Client*-SDOs (CSDO). Los parámetros de comunicación son obligatorios para todos los SDOs, aunque se pueden omitir si existe sólo un SDO.

2.2.2.5.3 Synchronisation Object (SYNC)

En una red CANopen, hay un dispositivo que es el productor de objetos SYNC y una serie de dispositivos consumidores de objetos SYNC. Cuando los consumidores reciben el mensaje del productor, abren su ventana de sincronismo y pueden ejecutar sus tareas síncronas.

El objeto de sincronización (SYNC) se emite de forma periódica por un nodo productor, se encarga de generar un reloj básico en el bus CANopen. Este mecanismo permite coherencia temporal y coordinación entre los dispositivos.

La transmisión de PDOs ligados al mensaje de sincronismo garantiza que los dispositivos sensores puedan muestrear variables de proceso de forma coordinada con las necesidades de los dispositivos actuadores sobre dichos procesos.

El COB-ID usado por este objeto de comunicación puede encontrarse en el índice 1005h del Diccionario de Objetos. Estos mensajes tienen un identificador perteneciente al grupo de alta prioridad, el 128. El campo de datos de del mensaje CAN de este objeto se envía vacío.

El comportamiento de estos mensajes está determinado por dos parámetros:

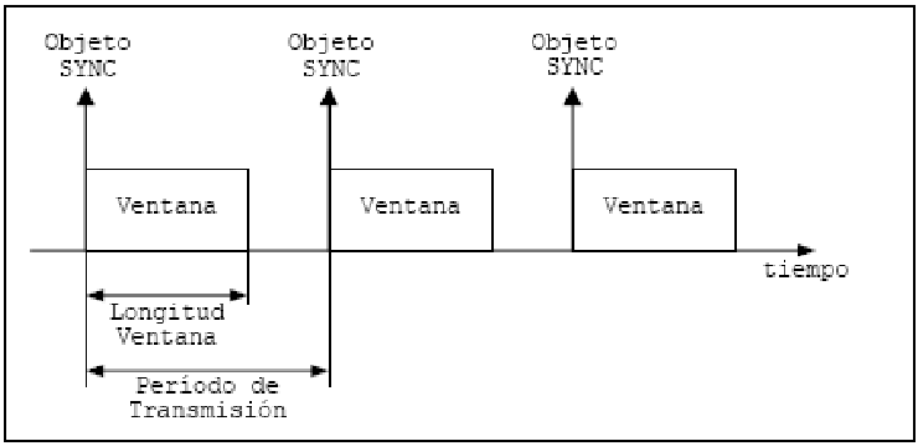


Figura 18: Parámetros de un objeto SYNC en CANopen.

Los PDOs síncronos son transmitidos dentro de un periodo de tiempo dado por la longitud de la ventana de sincronismo que está definido en el objeto 1007h. El periodo de ciclo de comunicación entre dos mensajes de sincronismo consecutivo está definido en el objeto 1006h. Estos dos valores se pueden cambiar en el proceso de configuración del Bus.

2.2.2.5.4 Time Stamp Object (TIME)

Este objeto representa el tiempo absoluto en "ms" (milisegundos) desde medianoche del 1 de enero de 1984. Proporciona a los dispositivos un tiempo de referencia común.

Es usado cuando el sistema requiere una alta precisión de sincronismo para ajustar la inevitable deriva de los relojes locales.

El identificador de dicho objeto, cuyo valor por defecto es el 256 (100h), está almacenado en el índice 1012h de Diccionario de Objetos.

2.2.2.5.5 Emergency Object (EMCY)

Estos mensajes se envían cuando ocurre un error interno en un dispositivo. Se transmiten al resto de dispositivos con la mayor prioridad. Pueden usarse como interrupciones o notificaciones de alertas.

Un mensaje de emergencia tiene 8 bytes. Su estructura se muestra a continuación:

COB-ID	Byte 0-1	Byte 2	Byte 3-7
080h + nodo ID	Emergency Error Code	Error Register (objeto 1001h)	Manufacture specific error field

Tabla 10: Estructura de un mensaje de emergencia CANopen.

- **Emergency Error Code** (2 bytes): código de error que causó la generación del mensaje de emergencia. Se trata de fallos internos de los dispositivos. La siguiente tabla nos muestra los códigos de error en hexadecimal:

Emergency Error Code	Meaning
00xx	Error Reset or No Error
10xx	Generic Error
20xx	Current
21xx	current, device input side
22xx	current, inside the device
23xx	current, device output side
30xx	Voltage
31xx	mains voltage
32xx	voltage inside the device
33xx	output voltage
40xx	Temperature
41xx	ambient temperature
42xx	device temperature
50xx	Device hardware
60xx	Device software
61xx	internal software
62xx	user software
63xx	data set
70xx	Additional modules
80xx	Monitoring
81xx	Communication
8110	CAN overrun
8120	Error Passive
8130	Life Guard Error or Heartbeat Error
8140	Recovered from Bus-Off
82xx	Protocol Error
8210	PDO not processed due to length error
8220	Length exceeded
90xx	External error
F0xx	Additional functions
FFxx	Device specific

'xx' es la parte dependiente del perfil del dispositivo

Tabla 11: Códigos de error para los mensajes de emergencia de CANopen.

- **Error Register** (1 byte): entrada con índice 1001h del Diccionario de Objetos. Cada bit de este registro indica una condición de error distinta cuando está a '1'. En la siguiente tabla se muestra el significado de cada bit:

Bit	Significado
0	Error genérico.
1	Problema de corriente.
2	Problema de tensión.
3	Problema de temperatura.
4	Error de comunicaciones.
5	Específico del perfil de dispositivo.
6	Reservado.
7	Específico del fabricante del dispositivo.

Tabla 12: Bits del Error Register de los mensajes de emergencia de CANopen.

- **Manufacturer-specific Error Field** (5 bytes): este campo puede usarse para información adicional sobre el error producido. Los datos incluidos y su formato se definen por el fabricante.

2.2.2.5.6 Network Management Object (NMT)

CANopen incluye una serie de mensajes para la administración y monitoreo de los dispositivos en la red. Estos están implementados en la capa CAL y reciben el nombre de servicios de gestión de red (*Network ManagemenT, NMT)*. Se trabaja con un modelo de comunicaciones **maestro-esclavo** en el cual un dispositivo es el maestro y el resto los esclavos.

Un dispositivo NMT esclavo puede encontrarse en alguno de los siguientes estados:

- ***Initialising:*** al encender el dispositivo se pasa directamente a este estado. Posteriormente a realizar las labores de inicialización el nodo transmite el mensaje *Boot-up* (único mensaje permitido) y pasa al estado *Pre-operational*.

COB-ID	Byte 0
700h + nodo ID	0

Tabla 13: Estructura del mensaje de Boot-up.

- ***Pre-operational:*** en este estado el dispositivo puede ser configurado mediante SDOs. En este estado se pueden enviar y recibir SDOs, mensajes de emergencia, de sincronización, *time Stamp* y mensajes NMT.
- ***Operational:*** el dispositivo ya ha sido configurado y funciona normalmente. Todos los objetos de comunicación están. En este estado se pueden enviar y recibir SDOs, mensajes de emergencia, de sincronización, *time Stamp*, mensajes NMT y enviar y recibir PDOs.
- ***Stopped:*** todos los objetos de comunicación dejan de estar activos. No se pueden enviar ni recibir PDOs ni SDOs, solo mensajes de NMT.

A continuación podemos ver en detalle el diagrama de los posibles estados de un nodo: (Ver figura 19).

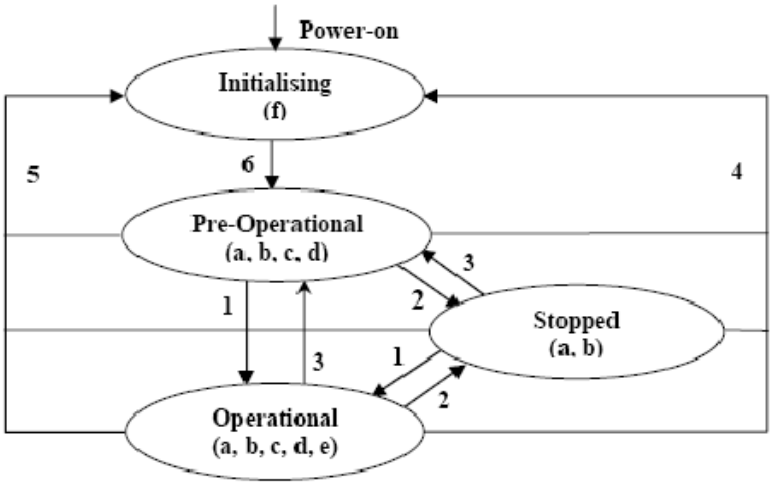


Figura 19: Diagramas de estados de un nodo en CANopen.

Transiciones entre estados (mensajes NMT):

1. *Start Remote Node*. (CS= 01h).
2. *Stop Remote Node*. (CS= 02h).
3. *Enter Pre*-Operational State. (CS= 80h).
4. *Reset Node*. (CS= 81h).
5. *Reset Communication*. (CS= 82h).
6. Inicialización terminada, entra en *Pre-Operational* directamente al mandar el mensaje *Boot-up*.

Sólo el NMT maestro puede enviar mensajes del módulo de control NMT, pero todos los esclavos deben dar soporte a los servicios del módulo de control NMT.

No hay respuesta para un mensaje NMT. Sólo los envía el maestro. Tienen el siguiente formato. (Ver Tabla 14).

COB-ID	Byte 0	Byte 1
000h	CS	Nodo ID

Tabla 14: Estructura de un mensaje NMT.

Con el Nodo ID (identificado de nodo) igual a cero, todos los NMT esclavos son direccionados (*broadcast*, significa que cualquier nodo perteneciente a la red puede escuchar todos los mensajes que se transmiten por la red).

El campo CS (*Command Specifier*) puede tener los valores mostrados en la Tabla 15.

Command Specifier	NMT Service
1	Start Remote Node
2	Stop Remote Node
128	Enter Pre-operational State
129	Reset Node
130	Reset Communication

Tabla 15: Valores del campo CS del mensaje NMT.

2.2.2.6 Estructura de un mensaje en CANopen

CANopen surge por la necesidad de definir como se asignan y utilizan los **identificadores** y los **datos** de los mensajes CAN:

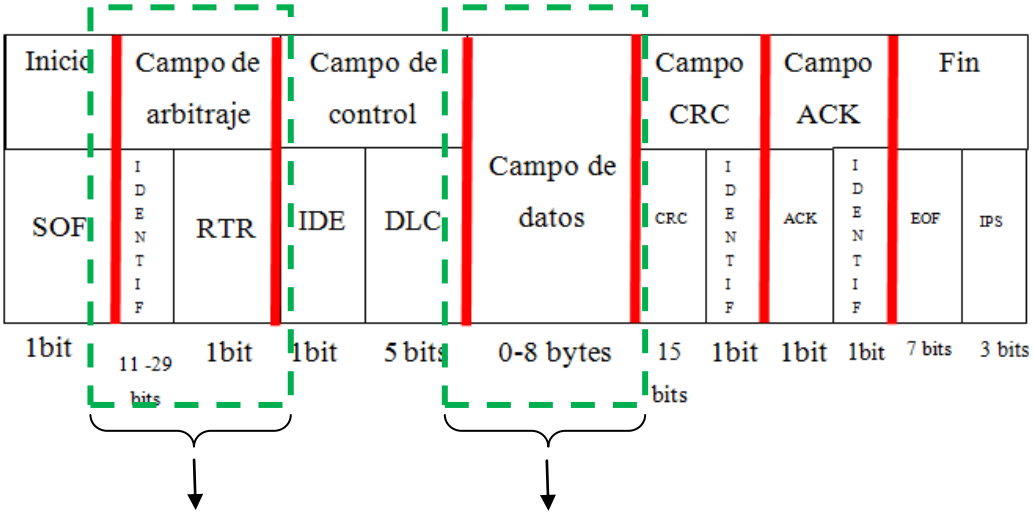


Figura 20: Estructura de un mensaje CANopen.

- **Identificador.** El identificador (COB-ID) es el responsable del arbitraje de los mensajes CAN y depende directamente del tipo de mensaje que se envia a través del bus. CANopen define la distribución de los identificadores de mensaje de manera que hay un mensaje de emergencia por nodo, mensajes de sincronización y *Time Stamp*, un SDO, mensajes NMT y cuatro PDOs de transmisión y cuatro de recepción por dispositivo.

  El identificador se compone de 11 bits, que se divide en dos campos: el primer campo es el código funcional, que está compuesto por los 4 bits más significativos y que determina la prioridad del objeto (SDO, PDO, SYNC, EMCY, etc.). El segundo campo es el Nodo-ID que está formado por los 7 bits menos significativos que permiten a un nodo maestro establecer la comunicación punto a punto con hasta 127 nodos esclavos.

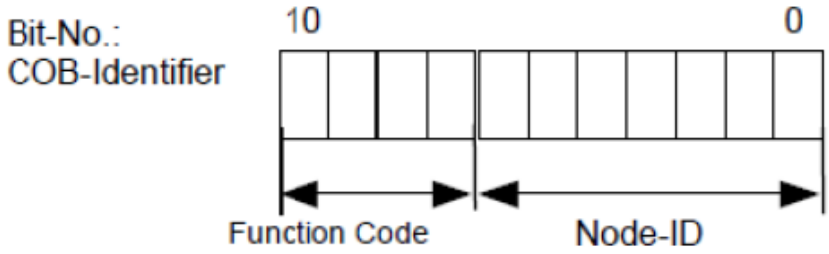


Figura 21: Estructura del identificador de mensajes CAN.

Broadcast objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID**	**Communication parameters at OD index**
NMT Module Control	0000	000h	–
SYNC	0001	080h	1005h, 1006h, 1007h
TIME STAMP	0010	100h	1012h, 1013h

Peer-to-Peer objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID** *	**Communication parameters at OD index**
EMERGENCY	0001	081h - 0FFh	1024h, 1015h
PDO 1 (transmit)	0011	181h - 1FFh	1800h
PDO 1 (receive)	0100	201h - 27Fh	1400h
PDO 2 (transmit)	0101	281h - 2FFh	1801h
PDO 2 (receive)	0110	301h - 37Fh	1401h
PDO 3 (transmit)	0111	381h - 3FFh	1802h
PDO 3 (receive)	1000	401h - 47Fh	1402h
PDO 4 (transmit)	1001	481h - 4FFh	1803h
PDO 4 (receive)	1010	501h - 57Fh	1403h
SDO (transmit/server)	1011	581h - 5FFh	1200h
SDO (receive/client)	1100	601h - 67Fh	1200h
NMT Error Control	1110	701h - 77Fh	1016h, 1017h

Tabla 16: Asignación de los identificadores de CAN en CANopen.

- **Datos.** Una vez se ha definido el identificador del mensaje, el esquema por defecto que tienen los 8 bytes de datos es el mostrado a continuación:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Tamaño de los parámetros	Índice		Subíndice	Parámetros			

Tabla 17: Estructura del campo de datos del mensaje CAN.

Tamaño de los parámetros. Indica el tamaño ocupado por el campo de parámetros. En la siguiente tabla se muestra los valores más comunes empleados:

Valor del campo "Tamaño de parámetros"	Tamaño del campo "Parámetros"
2Fh	8 bits
2Bh	16 bits
23h	32 bits
40h	0 bits

Tabla 18: Valores comunes del campo Tamaño de Parámetros.

Índice. Sirve para direccionar el objeto del Diccionario de Objetos con el que se quiere establecer la comunicación. (2 bytes => 16 bits).

Subíndice. Sirve para direccionar las sub-funciones del objeto señalado por el índice. (1 byte => 8 bits).

Parámetros. Son los datos que necesita el objeto del diccionario como datos de entrada. Su tamaño varía en función del tipo de dato que maneje el objeto. (32 bits máximos).

2.2.2.7 Diccionario de Objetos

La parte más importante del perfil de un dispositivo es la descripción del Diccionario de Objetos, que es un grupo de objetos accesibles de forma predefinida a través del bus mediante los cuales se realiza la comunicación con el dispositivo. Los objetos del Diccionario de Objetos están direccionados mediante un índice de 2 bytes y un subíndice de 1 byte. Su estructura es similar a una estructura en lenguaje C.

El Diccionario de Objetos viene explicado en el documento que se indica en la biografía (CANopen Programming) [6]. A continuación se va a mostrar un ejemplo de un objeto perteneciente al Diccionario de Objetos.

El objeto representado tiene el índice 1001h, que corresponde con el objeto *Error Register*.

Object description:

Index	1001_h
Name	Error register
Object code	VAR
Data type	UNSIGNED8

Entry description:

Access	RO
PDO mapping	No
Value range	UNSIGNED8
Default value	No

Bit	Description
0	Generic error
1	Current
2	Voltage
3	Temperature
4	Communication error
5	Device profile specific
6	Reserved (always 0)
7	Manufacturer specific.

Figura 22: Ejemplo de objeto del Diccionario de Objetos.

En la figura anterior, se proporciona información relacionada con el objeto *Error Register*. Indica el nombre, el índice, el tipo de dato, el tipo de acceso (RO- lectura; WO- escritura; RW- lectura y escritura). También proporciona información a sobre si el objeto se puede hacer mapeado.

Otra información que se muestra en el Diccionario de Objetos, es la descripción del campo de datos.

2.2.2.8 Máquina de estados

La máquina de estados del dispositivo describe el estado de la unidad y las posibles secuencias de control de la unidad. El estado de la unidad puede ser cambiada por medio del objeto ***Control Word*** y acordar los acontecimientos internos. El estado corriente de la unidad es reflejado en el objeto ***Status Word***.

La máquina de estados mostrada a continuación, describe la máquina de estados de la unidad con respecto al control de la etapa de potencia, como resultado de los comandos del usuario y fallos internos de la unidad.

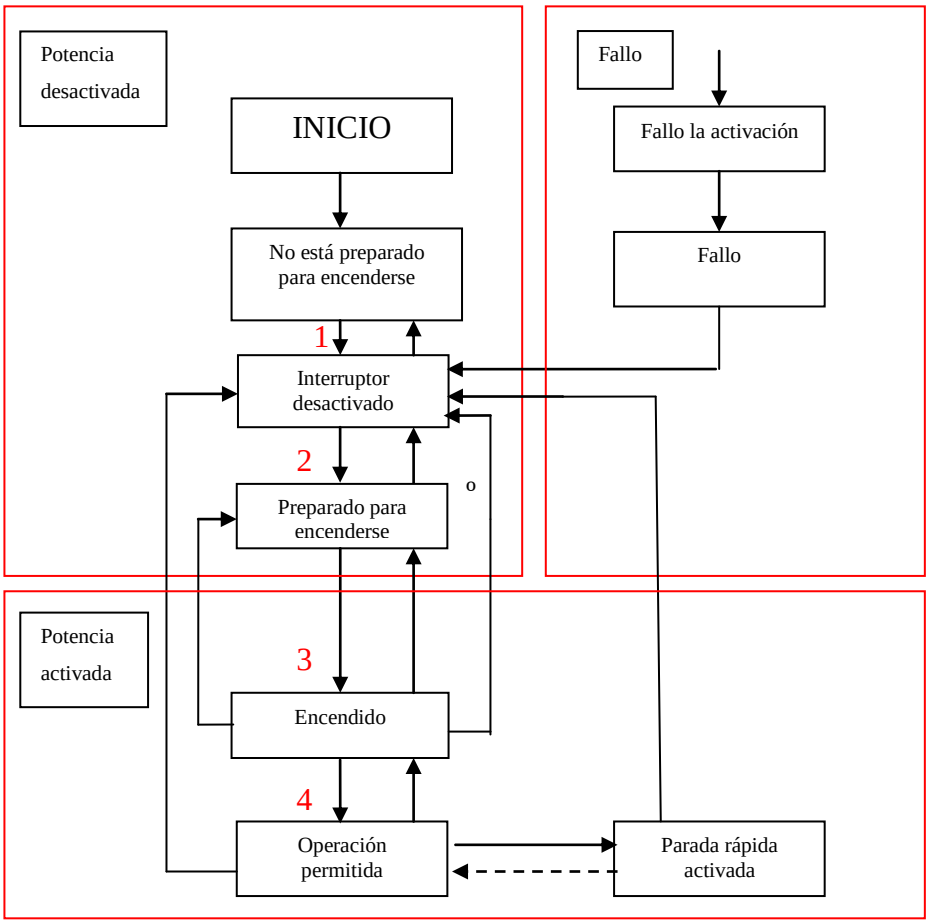


Figura 23: Maquina de estados de la unidad. Estados y transiciones.

Ahora vamos a indicar las pasos a seguir para realizar la inicialización de un nodo del driver. Para esto utilizamos los siguientes mensajes:

1) Enviamos un mensajes NMT al nodo 1, con la orden para iniciar el nodo 1, para alcanzar el estado de interruptor desactivado, con los siguientes datos:

COB-ID	00h
Data	01 01

Tabla 19: Mensaje NMT para iniciar el nodo 1.

2) Cambiamos el estado del driver a preparado para encenderse. Para ello enviamos un mensaje del tipo SDO-Receive (COB-ID 600h + nodo ID).

COB-ID	601
Data	2B 40 60 00 06 00 00 00

Tabla 20: Transición a estado preparado para encenderse.

3) Mandamos el siguiente mensaje para cambiar el estado del driver a encendido. Para ello enviamos un mensaje de tipo SDO-Receive (COB-ID 600h + nodo ID).

COB-ID	601
Data	2B 40 60 00 07 00 00 00

Tabla 21: Transición a estado encendido.

4) Con el siguiente estado cambiamos el estado del driver a operación permitida. Para ello enviamos un mensaje de tipo SDO-Receive (COB-Id 600h + nodo ID).

COB-ID	601
Data	2B 40 60 00 0F 00 00 00

Tabla 22: Transición a estado operación permitida.

Estos tres últimos mensajes se podrían haber enviado a través de PDOs (COB-ID 200h + nodo ID). Para este Trabajo Fin de Grado se utilizan dos objetos de comunicación que son: PDO-1 *Receive* (del 201h al 223h) y SDO-*Receive* (del 601h al 623h). PDO-1 *Receive* tiene mapeado por defecto el objeto 6040h *Control Word.*

A partir de este punto, el driver ya esta inicializado, solo faltaría elegir el modo de operación necesario y enviar las posiciones y velocidades necesarias. Esto se verá más adelante, en la explicación de la aplicación.

2.2.2.9 Objetos de control y estado del driver

Hay ciertos objetos del Diccionario de Objetos que tienen el objetivo de controlar y de comprobar el estado de los actuadores que controla el driver, así como seleccionar los diferentes modos de operación del mismo.
Entre los objetos más importantes vamos a destacar los tres siguientes:

- ***Control Word:*** cambia los estados del dispositivo.
- ***Status Word*:** verifica el estado actual del driver.
- ***Mode of Operation:*** cambia los diferentes modos de operación del driver.

2.2.2.9.1 Control Word

El índice de este objeto corresponde con el 6040h. Este objeto se utiliza para controlar la máquina de estados del driver, sirviendo de transición entre estados. Su principal función es la de habilitar/deshabilitar la alimentación del driver, así como para parar/comenzar los movimientos, también se puede usar para sacar del estado de fallo al driver. La *Control Word* consta de 16 bits para poder llevar a cabo las funciones de dicho objeto. En la siguiente tabla se muestra la descripción de cada bit.

Bit	Valor	Descripción
15	0	Modo registro inactivo.
	1	Modo registro activo.
14	1	Cuando se produzca una actualización, no se actualizan los valores de posición y velocidad.
13	1	Se cancela la ejecución de la función de TML llamada por el objeto 2006h. El bit es reinicializado por el driver cuando se ejecuta la orden.
12	0	Sin acción.
	1	Si el bit 14= 1 pone la posición demandada a 0. Si el bit 14= 0 pone la posición actual a 0.
11		El funcionamiento de este bit depende del funcionamiento del driver.

Bit	Valor	Descripción
10		Reservado.
9		Reservado.
8	0	Sin acción.
	1	El motor frena.
7	0	Sin acción.
	1	Reseteo de los fallos. En la transición de 1 a 0 de ese bit se resetean los fallos.
4,5,6		Estos bits dependen del modo de operación. Según el modo en que se encuentre el driver, el valor de estos bits tiene un sentido u otro. (Se explicara con mayor detalle más adelante).
3		Habilitar operación.
2		Parada rápida.
1		Habilitar alimentación del driver.
0		Encendido.

Tabla 23: Control Word.

2.2.2.9.2 Status Word

El índice de este objeto corresponde con el objeto 6041h. Con este objeto se evalúa el estado actual en el que se encuentra el driver. La *Status Word* consta de 16 bits para poder llevar a cabo las funciones de dicho objeto. En la siguiente tabla se muestra la descripción de cada bit.

Bit	Valor	Descripción
15	0	Eje apagado. Etapa de potencia apagada. No se produce control del motor.
	1	Eje encendido. Etapa de potencia desactivada. Se realiza control del motor.
14	1	El último evento se ha producido.
	0	El evento no se ha ajustado o no se ha producido, todavía.
13-12		Especifico del modo de operación. El significado de este bit se

Bit	Valor	Descripción
13-12		Mostrara en los apartados de desarrollo de la aplicación.
11		Limite interno activado.
10		Objetivo alcanzado.
9	0	Remoto- El drive esta en modo local y no se ejecutara el mensaje.
	1	Remoto- Los parámetros del driver pueden ser modificados vía CAN y el driver ejecutará el mensaje.
8	0	No se ejecutara una función TML o un *homing*. La ejecución de la última función TML o *homing* se ha completado.
	1	Una función TML o *homing* se ejecuta. No se puede llamar a otra función TML o *homing*, hasta que no se finalice o aborte la función TML o *homing* que se está ejecutando.
7	0	Sin peligro.
	1	Peligro. Una función TML / *homing* es llamada, mientras otra función TML o *homing* está todavía en ejecución. La última llamada será ignorada.
6		Encendido deshabilitado.
5		Parada rápida. Cuando este bit está a 0, el driver realizará una parada rápida.
4	0	La tensión de alimentación está presente en el driver.
	1	La tensión de alimentación no está presente en el driver.
3		Fallo. Si está a 1 el bit, un fallo está o estaba presente en el driver.
2		Operación habilitada.
1		Encendido.

Tabla 24: Status Word.

2.2.2.9.3 Mode of Operation

El índice de este objeto corresponde con el 6060h. Este objeto se utiliza para elegir el modo de control del driver. Es un objeto de solo escritura, por lo que no se puede leer de él. El *Mode of Operation* no consta de 16 bits para poder llevar a cabo las funciones de dicho objeto, sino que según el valor del número entero contenido en el índice, se elige un modo u otro de funcionamiento. En la siguiente tabla se muestra la descripción de cada valor.

Valor	Descripción
-128 hasta el -6	Reservado.
-5	Modo de momento de rotación de referencia externo.
-4	Modo de velocidad de referencia externa.
-3	Modo de posición de referencia externa.
-2	Modo de posición ECAM.
-1	Modo de posición EG.
0	Reservado.
1	Modo de perfil de posición.
2	Reservado.
3	Modo de perfil de velocidad.
4	Reservado.
5	Reservado.
6	Modo homing.
7	Modo de posición interpolado.
8 hasta 127	Reservado.

Tabla 25: Mode of Operation.

Por medio de estos tres objetos descritos en el apartado 2.2.2.9, hemos desarrollado el Trabajo de Fin de Grado. A continuación en el capitulo tres vamos a describir los elementos de hardware empleados.

Capítulo 3: Elementos hardware empleados

3.1 Introducción

Una vez se han explicado en el capitulo anterior las herramientas de software empleadas, ahora procederemos a describir los sistemas de hardware que hemos utilizado para implementar la aplicación desarrollada en este Trabajo de Fin de Grado.

Este capítulo tiene el objetivo de hacer una idea de los componentes básicos del sistema, explicando cuales son las principales características de cada dispositivo. Las conexiones y otros datos relacionados con el montaje de los dispositivos será explicado en el siguiente capítulo.

Los sistemas de hardware empleados, son los que se enumeran a continuación:

- **Soporte del robot ASIBOT:** estructura mecánica que soporta el brazo robótico ASIBOT. Este elemento, es el encargado de dar soporte y trasladar a la posición requerida el brazo robótico ASIBOT.
- **Actuadores y sensores:** sistema 322951 de Maxon, que está formado por:
    - El motor brushed RE30 de referencia 310009.
    - El encoder relativo MR de 500 ppv de referencia 228452.
- **Control de motores:** driver ISCM8005 de Technosoft.
- **Comunicación:** tarjeta HICO.CAN-miniPCI de Emtrion.
- **CPU:** RoBoard RB-100 de DMP Electronics Inc.

3.2 Soporte del robot asistencial ASIBOT

Dado que el robot ASIBOT [7] se encuentra todavía en fase de desarrollo, el diseño realizado es para controlar la estructura mecánica de un grado de libertad. El elemento que soportaría el robot es el que se muestra en la figura siguiente:

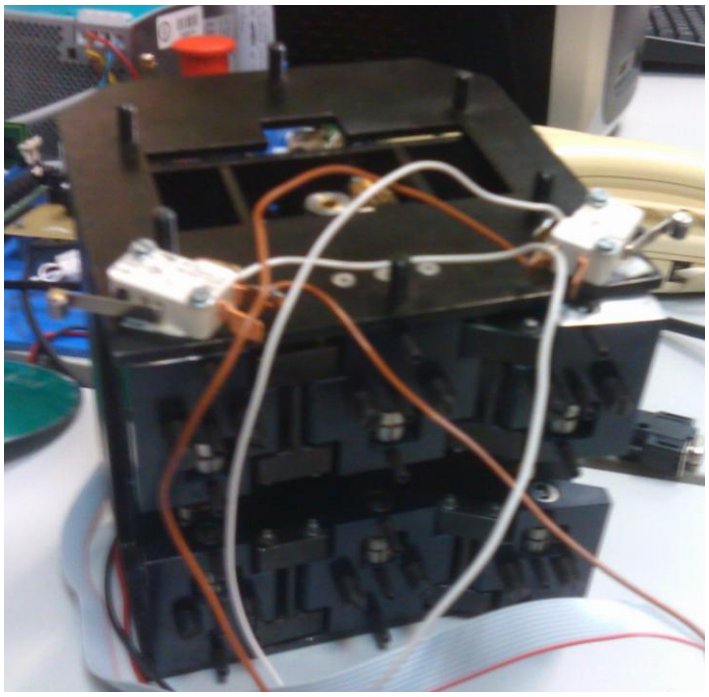


Figura 24: Soporte del robot asistencia ASIBOT.

Este elemento se encarga de soportar el robot ASIBOT. En su interior se encuentra alojado el sistema 322951 de Maxon, como se describió en la introducción de este capítulo, dicho sistema de está formado por un motor DC brushed y un encoder.

Por lo tanto, el soporte se encargará de situar el robot ASIBOT en la posición deseada, disponiendo únicamente de un motor y un encoder. Además para satisfacer las necesidades de potencia y par, el motor cuenta con un sistema de reducción.

Dicho soporte se montará sobre una cremallera que se ha montado alrededor de la silla de ruedas. La situación de la cremallera se muestra en las siguientes figuras:

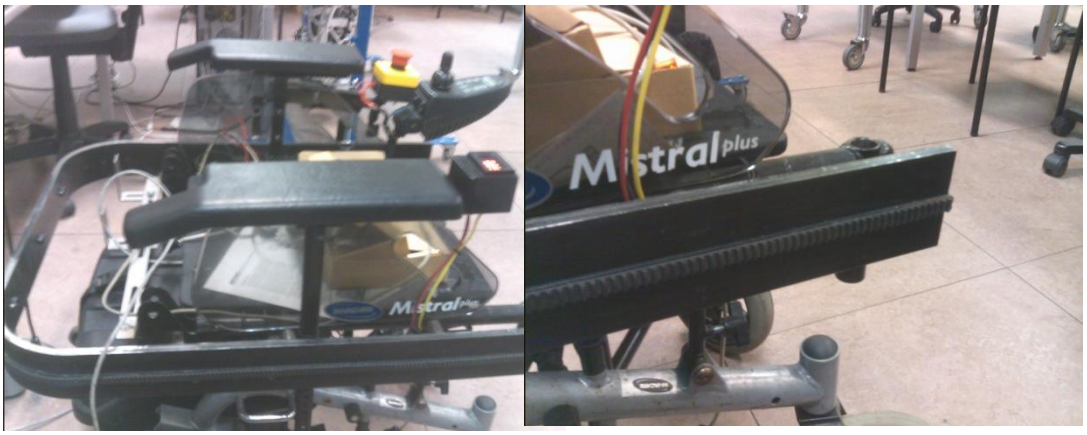


Figura 25: Localización de la cremallera en la silla de ruedas.

3.3 Actuadores y sensores: sistema 322951 de Maxon

En la robótica uno de los elementos más importantes son los elementos de campo, que son los sistemas actuadores y sensores. Estos sistemas ayudan al robot a percibir su entorno y poder actuar sobre él. En el caso de este Trabajo de Fin de Grado, los elementos de campo consisten en un motor *brushed* y en un sensor de posición angular.

Los actuadores son dispositivos mecánicos cuya función principal es proporcionar fuerza para mover otro dispositivo mecánico. La fuerza que provoca el actuador puede ser de tres fuentes: presión neumática, presión hidráulica y fuerza motriz eléctrica. En este caso se han utilizado actuadores de tipo eléctrico.

Un sensor es un dispositivo encargado de detectar magnitudes físicas o químicas (llamadas magnitudes de instrumentación) y transformarlas en magnitudes eléctricas. Siendo la magnitud física medida de tipo desplazamiento circular o rotacional. En este caso se ha utilizado un encoder. El encoder es un elemento electromecánico usado para convertir la posición angular de un eje en un código digital.

Para el desarrollo de este Trabajo de Fin de Grado se ha utilizado un sistema 322951 de Maxon [8]. Este sistema incorpora en un único bloque un dispositivo actuador (motor brushed RE30 de referencia 310009) y un dispositivo sensor (encoder relativo

MR de 500 ppv de referencia 228452). De esta forma se consigue mayor robustez y fiabilidad, además de facilitar su integración en el soporte del robot.

A continuación, se muestra la Figura 26 con la intención de facilitar la comprensión del sistema completo.

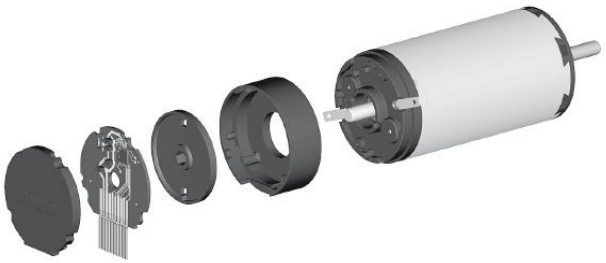


Figura 26: Sistema 322951 de Maxon.

3.3.1 Motor brushed (310009)

Este motor pertenece al programa RE de motores de Maxon de corriente continua (DC) que está equipado con potentes imanes permanentes. Su diámetro es de 30 mm, sus escobillas son de grafito y con una potencia equivalente de 60W.

A continuación se muestra una figura ilustrativa de las partes más importantes de los motores utilizados en la aplicación.

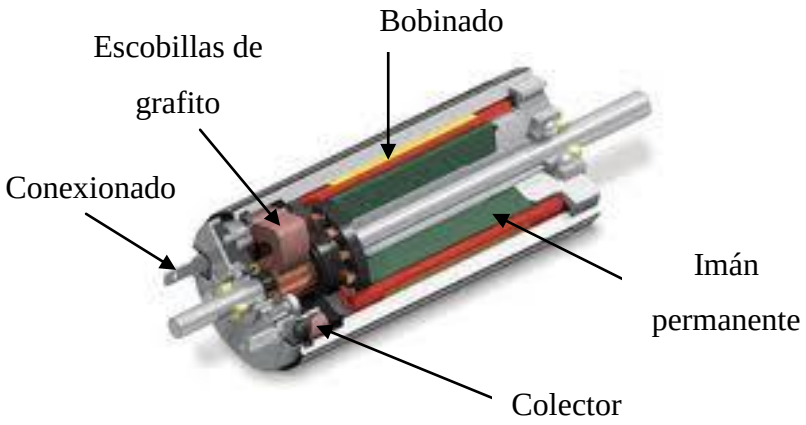


Figura 27: Motor RE 30 de Maxon.

La siguiente tabla muestra las características más importantes proporcionadas por la hoja de características del fabricante.

Valores a tensión nominal		Características	
Descripción	**Valor**	**Descripción**	**Valor**
Voltaje nominal [V]	48	Resistencia en bornes [Ω]	2,52
Velocidad en vacío [rpm]	8490	Inductancia en bornes [mH]	0,513
Corriente en vacío [mA]	78,5	Constante de par [mNm/A]	53,8
Velocidad nominal [rpm]	7750	Constante de velocidad [rpm/V]	178
Par nominal [mNm]	88,2	Relación velocidad par [rpm/mNm]	8,33
Corriente nominal [A]	1,72	Constante de tiempo mecánica [ms]	3,01
Par de arranque [mNm]	1020	Inercia del motor [gcm^2]	34,5
Corriente de arranque [A]	19		
Máximo rendimiento [%]	88		

Tabla 26: Características del motor RE30 de Maxon.

3.3.2 Encoder relativo (228452)

Este encoder relativo es de tipo magnético-resistivo (MR) ya que posee un disco magnético multipolar montado en el eje del motor que genera una variación de tensión sinusoidal en el sensor MR.

En la siguiente figura se muestra el interior de este tipo de encoders, además se señalan las partes más importantes que componen el encoder.

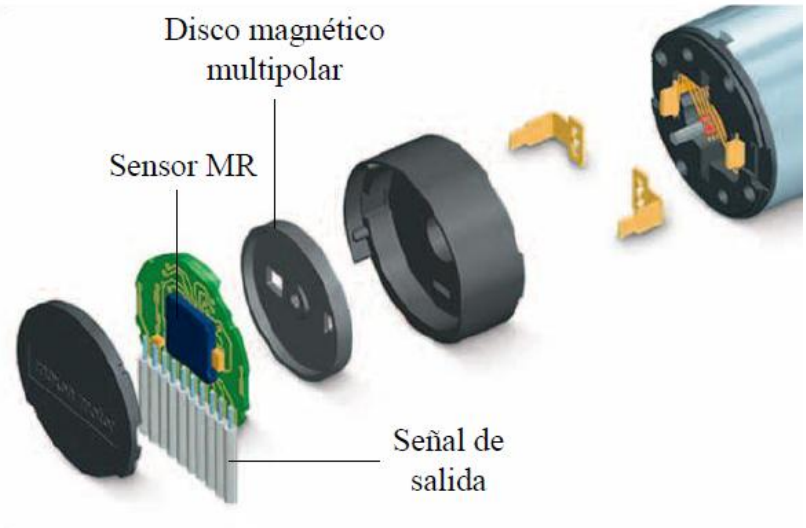


Figura 28: Encoder relativo MR

En la tabla mostrada a continuación se presentan las principales características de este tipo de encoder, facilitadas por el fabricante.

Características	Valor
Numero de pulsos por vuelta	500
Número de canales	3
Máx. Frecuencia de funcionamiento [kHz]	200
Máx. Velocidad [rpm]	24000
Tensión de alimentación [V]	5 ± 0.05
Señal de salida	TTL compatible
Desfase [º]	90ºe ± 45ºe
Anchura de pulso índice [º]	90ºe ± 45ºe
Rango de temperaturas [ºC]	-25: + 85
Momento de inercia de rueda de código [gcm^2]	1,7
Corriente máx. de salida por canal [mA]	5

Tabla 27: Características encoder relativo MR.

3.4 Control de motores: driver ISCM8005 de Technosoft

Introducción

Ahora se procederá a describir el driver utilizado para el control del único motor presente en el soporte del robot ASIBOT. Un driver es un dispositivo cuya finalidad principal es la regulación del flujo de intensidad que recibe un motor en función de de la lectura obtenida a través de un sensor y de las órdenes que reciba de la aplicación que lo controla. A continuación se muestra una imagen del esquema de control.

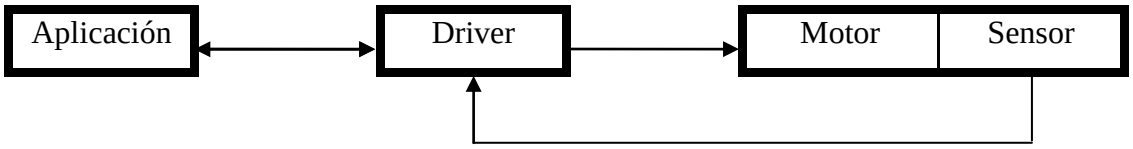


Figura 29: Esquema de control.

Este tipo de dispositivos incluyen reguladores para que sea posible comandar el motor con una referencia de posición, velocidad o par. Los reguladores que suelen incluir son de tipo PID (*Proportional-Integrative-Derivative)* que son los reguladores más básicos y efectivos.

El dispositivo utilizado para este propósito es la tarjeta ISCM8005 desarrollado por la empresa Technosoft [9], mostrada en la Figura 30.

Figura 30: Driver ISCM8005.

Este dispositivo pertenece a una familia de servo variadores totalmente digitales inteligentes, basados en las últimas tecnologías DSP. Estos driver ofrecen un gran rendimiento de la unidad que se combina con un controlador integrado de movimiento.

Este driver es adecuado para el control de motores de corriente continua sin escobillas, de motores de corriente alterna sin escobillas, de motores de corriente continua con escobillas y de motores de paso a paso.

El ISCM8005 acepta encoders de realimentación, tanto encoders incrementales como sensores de efecto Hall.

Todos los ISCM8005 pueden realizar un control de posición, control de velocidad o control del par. Y puede trabajar en un eje, múltiples ejes o en configuraciones independientemente.

Gracias al control de movimiento integrado que incorpora dicho driver, en los ISCM8005 se combinan el amplificador y controlador de movimiento con funciones de PLC en una sola unidad compacta, siendo capaz de ejecutar movimientos complejos sin la necesidad de la intervención de un controlador de movimiento externo.

Por medio del uso del lenguaje Technosoft Motion Language (TML) se pueden ejecutar en el driver las siguientes acciones:

- Configurar los modos de movimiento (varios tipos de perfiles, PVT, PT, etc.).
- Cambiar los modos de movimiento y / o los parámetros de movimiento.
- Ejecución de secuencias *homing.*
- Se controla el flujo del programa a través de las siguientes acciones:
    - Saltos condicionales y llamadas a funciones TML.
    - *Interrupts* (interrupciones) TML predefinidas o condiciones programadas (protecciones activas, transiciones de final de carrera o la captura de entradas, etc.).
    - Esperas para que los eventos programados se produzcan.

- Gestión de entradas / salidas digitales y señales de entrada analógicas.
- Ejecución de operaciones aritméticas y lógicas.
- Realizar transferencia de datos entre ejes.
- Control de movimiento de un eje por otro a través de comandos de movimiento que se envían entre ejes.
- El envío de comandos a un grupo de ejes (multicast). Esto incluye la posibilidad de iniciar al mismo tiempo las secuencias de movimiento en todos los ejes del grupo.
- Sincronizar todos los ejes de una red.

Características principales

- Herramientas software EasyMotion y EasySetup para la configuración y programación de los drivers.
- Unidades digitales para el control de motores de corriente continua (DC) sin escobillas, motores de corriente alterna (AC) sin escobillas, motores de corriente continua (DC) con escobillas y motores paso a paso, con una función de controlador y lenguaje de movimiento de alto nivel TML.
- Control de posición, de velocidad o de par.
- Varios modos de programación de movimiento:
    - Perfiles de posición con forma trapezoidal o tipo curva-S.
    - Posición, velocidad, tiempo (PVT) 3^{rd} orden de interpolación.
    - Posición, tiempo (PT) 1^{st} orden de interpolación.
    - Referencia externa analógica o digital.
    - 33 modos *homing.*
- Interfaz para encoder incremental, diferencial o de colector abierto.
- Interfaz para encoder de efecto Hall.
- 7 líneas dedicadas a entradas-salidas digitales.
    - 3 líneas de entradas-salidas digitales.
    - 2 líneas de entradas-salidas digitales compartidas con 2 entradas analógicas (0…3 V).
    - Entrada *RESET*.

- Entrada de emergencia para apagar el dispositivo en caso de fallo (*ENABLE).*

- Interfaz serie RS-232 (hasta 115200 bps).
- CAN-Bus 2.0B hasta 1Mbit/s, con protocolo de comunicación:
  - CANopen compatible con los estándares CiA: DS301 y DSP402.
  - TMLCAN compatible con todos los drivers Technosoft con interfaz CAN-Bus.
- Memoria interna SRAM de 1,5K x 16.
- Memoria EEPROM para almacenar programas en TML y datos de configuración.
- Hardware de selección de *Axis* ID (identificador de eje).
- Señal de PWM de frecuencia de conmutación de 10 kHz.
- Alimentación de la red puede variar de 12 a 48 V (típico 24 V).
- Alimentación del motor de 5 a 80 V; intensidad nominal 5 A; intensidad de pico 16 A (intensidad en el arranque).
- Mínima inductancia de carga: 50 µH a 12 V; 200 µH a 48 V; 330 µH a 80 V.
- Temperatura ambiente de operación puede variar de 0 a 40 ºC.
- Protección hardware frente a sobreintensidades.

Estructura del driver

Este driver cuenta con dos bloque diferenciados, uno en cada cara del driver: Cara A y la Cara B. En la Figura 31 se muestra la Cara A y en la Figura 32 la Cara B.

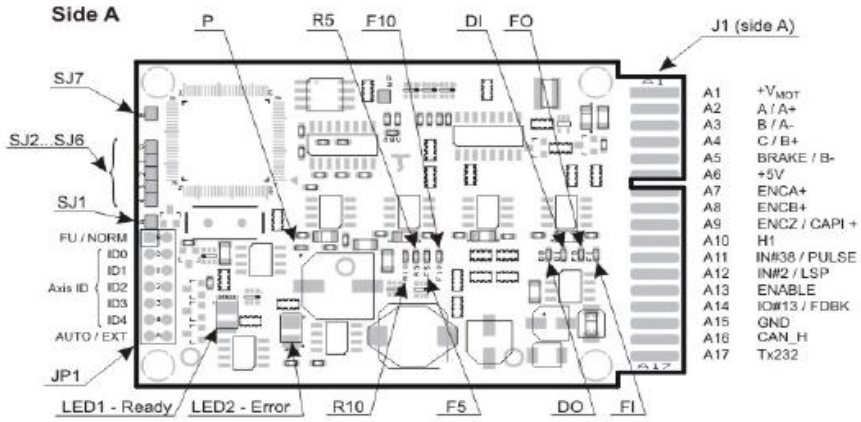


Figura 31: Cara A del driver ISCM8005.

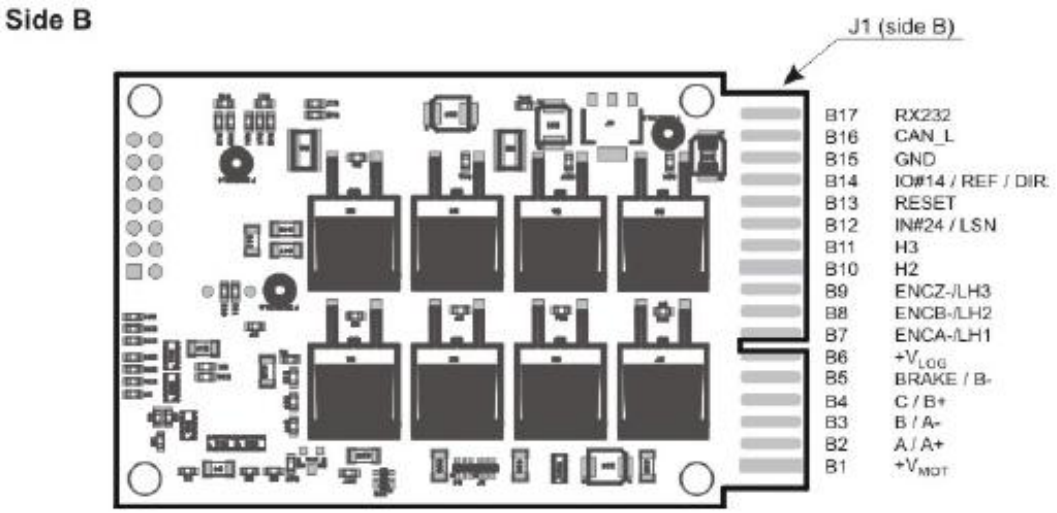


Figura 32: Cara B del driver ISCM8005.

- **Cara A:** en esta cara se encuentra toda la electrónica digital encargada de procesar el control, la comunicación, el almacenaje de configuraciones y programas.
- **Cara B:** en esta cara se encuentra toda la electrónica de potencia encargada de alimentar los motores.

Unidades internas del driver

Este dispositivo cuenta sus propias unidades internas, por lo que es necesario pasar los grados y las revoluciones por minuto a las unidades internas del sistema.

Es importante destacar que los valores en unidades internas deben de ser convertidos a hexadecimal. Una vez hecho esto introducimos el valor obtenido en dentro del campo de datos del mensaje CAN, concretamente en el área de "parámetros" del campo de datos (desde el byte 4 al byte 7). (Ver Figura 33 y Tabla 28 para facilitar la comprensión).

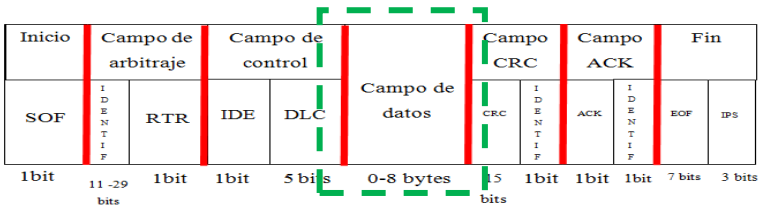


Figura 33: Mensaje CAN (remarcado el Campo de datos).

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Tamaño de los parámetros	Índice		Subíndice	Parámetros			

Tabla 28: Estructura del campo de datos de un mensaje CAN (remarcado los parámetros).

Para pasar de **GRADOS** [SI] a **UNIDADES INTERNAS** [IU]:

$$Posición\ motor\ [IU] = \frac{4\ \cdot T_r \cdot N \cdot POS[SI]}{360}$$

T_r = Relación de transmisión entre la carga y el motor (1:250).

N = Nº de líneas del encoder. (500 líneas en nuestro caso).

POS [SI] = Grados a convertir en unidades internas.

Posición motor [IU] = Posición en unidades internas.

Para pasar de **RPM** (revoluciones por minuto) [SI] a **UNIDADES INTERNAS** [IU]:

$$Velocidad\ motor\ [IU] = (\frac{4\ \cdot T_r \cdot N \cdot VEL[SI] \cdot T}{60}) \cdot 65536$$

T_r = Relación de transmisión entre la carga y el motor (1:250).

N = Nº de líneas del encoder. (500 líneas en nuestro caso).

VEL [SI] = Revoluciones por minuto a convertir en unidades internas.

Velocidad motor [IU] = Velocidad en unidades internas.

T = 0,001 s → 1 ms.

3.5 Comunicación: tarjeta HICO.CAN-miniPCI de Emtrion

Introducción

Para el desarrollo de este Trabajo de Fin de Grado disponemos de la tarjeta de comunicación HICO.CAN-miniPCI [10]. La gran ventaja que tiene dicho elemento es su reducido tamaño, por lo que se puede integrar dentro del soporte del robot asistencial ASIBOT con gran facilidad. Se utiliza dicho elemento debido a que la CPU RoBoard RB-100, presenta un puerto PCI mini, y por este motivo se decidió implementar el sistema con una tarjeta de comunicación con interfaz mini PCI.

A continuación se muestra en la Figura 34 una imagen de la tarjeta de comunicación HICO.CAN-miniPCI utilizada para este Trabajo de Fin de Grado.

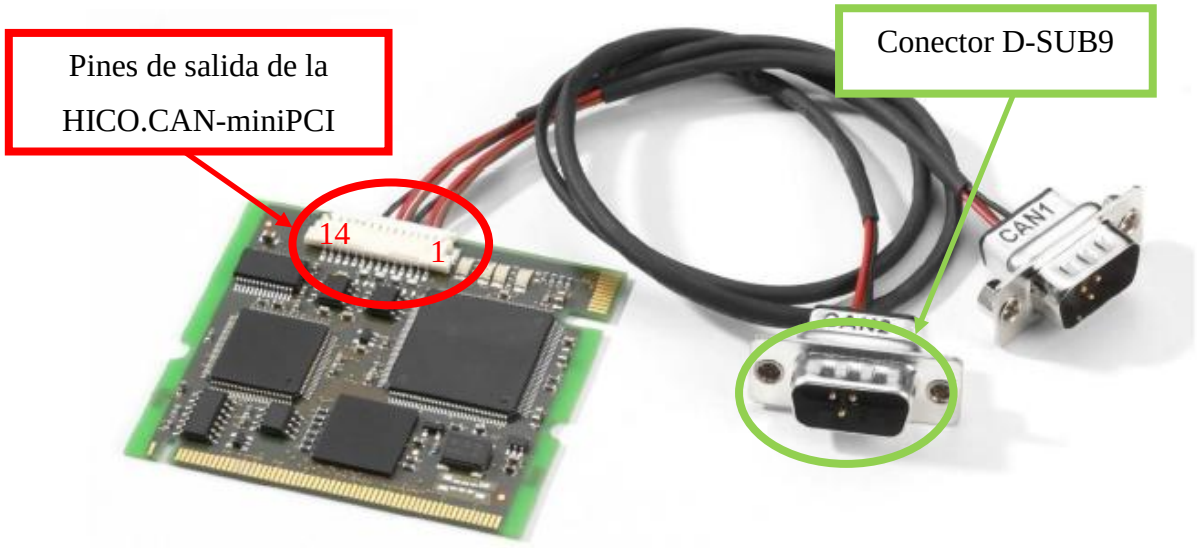


Figura 34: HICO.CAN-miniPCI.

El controlador se implementa como un módulo de núcleo cargable (*Loadable Kernel Module, LKM*), siendo capaz de controlar múltiples tarjetas y múltiples clientes (aplicaciones) simultáneamente en el sistema. La interfaz para el controlador es *stream* (interfaz de dispositivo de caracteres) que ya se ha establecido como la interfaz estándar de bus CAN en Linux. La función del controlador API sigue el estándar POSIX en la medida de lo posible.

El trafico CAN puede ser leído y escrito, con las llamadas a las funciones del sistema *read ()* y *write ()* y por medio del uso de comandos específicos de hardware (como el ajuste de velocidad de bits, por ejemplo) que se realiza a través de las llamadas Ioctl ().

Los pines de salida de la tarjeta HICO.CAN-miniPCI se muestran en la siguiente tabla:

PIN	SEÑAL	D-SUB9
1	3,3 V	
2	CAN1_H	CANAL 1
3	CAN1_L	CANAL 1
4	CAN1 ERR	
5	GND	CANAL 1
6	CAN2_H	CANAL 2
7	CAN2_L	CANAL 2
8	CAN2 ERR	
9	GND	CANAL 2
10	No usado	
11	No usado	
12	No usado	
13	No usado	
14	No usado	

Tabla 29: Pines de salida para la tarjeta de comunicación HICO.CAN-miniPCI.

Mediante dos conectores de D-SUB9 (uno para cada canal) se conecta la tarjeta a la red CAN. Los pines de dichos conectores están definidos acordes con las recomendaciones de CiA, mostrado en la siguiente figura:

Figura 35: Conector D-SUB9.

PIN	SEÑAL
2	CAN_L
3	GND
7	CAN_H

Tabla 30: Asignación de los pines del conector D-SUB9.

El bus CAN debe llevar elementos terminadores entre los pines CAN_H y CAN_L en los extremos del bus. Esto se puede realizar directamente sobre el cable.

Características principales

- Tarjeta de interfaz miniPCI tipo IIIA.
- Dispone de dos canales de CAN: CAN 2.0A y CAN 2.0B.
- Controlador Philips LPC2292 (ARM7), 60 MHz.
- Transceptores CAN integrados con señales TTL para aislamiento galvánico opcionales.
- Opción de personalizar el *firmware* por medio de 32 bits integrados.
- Soporta todos los estantadares (CIA-DS-102) con un 100% de velocidad de transmisión.
- *Timestamp* de 1µs de resolución.
- *Bufeer* interno de capacidad de 500 mensajes CAN por nodo.
- Opcional: rango de temperatura de -40 ºC hasta 85 ºC, aislamiento galvánico de 3000 V.
- El dispositivo soporta Windows CE, Windows XP, Windows 98/ME, Windows 2000, Windows NT, Linux 2.6 y QNX 6.3.

- Con un adaptador el módulo se puede usar en un bus normal PCI, PC/104+ o PCI-104.

Características de hardware

- **Dos nodos CAN independientes**: (CAN 2.0B) con transceptores integrados.
- **Dimensiones:** 60 mm x 51 mm x 3 mm. Compatible con las especificaciones miniPCI tipo IIIA.
- **Peso**: la tarjeta tiene un peso de 11 g y 27 g con los cables.
- **Cable:** conector 1x 14 pines; 2x DSUB9.
- **Consumo de corriente:** 80ma (max).
- **Rango de temperaturas de funcionamiento:** estándar 0 ºC a 70 ºC; industrial de -40 ºC a 85ºC.
- **Leds de estado para errores de CAN bus y para el tráfico CAN bus.**

Características de operación

- **Detección automática de la velocidad de transmisión:** el tráfico del Bus es escuchado en modo pasivo y analizado para escoger la tasa de Baudios correcta.
- **Modo activo y modo pasivo:** en modo pasivo, el tráfico del bus CAN es escuchado, pero esto no afecta a la monitorización (*monitoring*).
- **Mensaje flexible:** filtrando con hasta cuatro filtros por nodo, los filtros son de tipo código y mascara o ID-*range*.
- **Buffer de mensajes:** 500 mensajes CAN por cada nodo.
- ***Timestamp*:** con resolución de 1 µs que permiten una reproducción más exacta del flujo de datos.
- **Velocidad de transmisión:** soporta todos los estándares de CIA-DS-102. Las velocidades de transmisión disponibles son 10 Kb/s, 20 Kb/s, 50 Kb/s, 125 Kb/s, 250 Kb/s, 500 Kb/s, 800 Kb/s y 1 Mb/s.

Linux 2.6 Driver

El controlador proporciona una interfaz para el sistema de archivos de estándar POSIX para la tarjeta – *open (), read (), write ().* Comandos especiales (como establecer la velocidad de transmisión del bus en baudios) que se realiza mediante llamadas Ioctl. El siguiente fragmento de código muestra un ejemplo trivial de uso.

```
int fd,rate;
struct can_msg msg;

/* open CAN node 0 for reading and writing */
fd = open( "/dev/hicocan0" ,O_RDWR);

/* Set baudrate to 20kbit/s */
rate=BITRATE_500k;
ioctl(fd, SET_BITRATE, &rate);

/* Start to read messages */
while(running)
{
        read(fd,&msg,sizeof(msg));
        printf( "received message with id %x",msg.id);
}

/* Stop using the node */
close(fd)
```

3.6 CPU: RoBoard RB-100 de DMP Electronics Inc

Introducción

El primer requisito lo impone el reducido espacio disponible para alojar los elementos que forman el sistema de control de motores, que está compuesto por: el driver ISCM8005, la tarjeta de comunicación HICO.CAN-miniPCI y la CPU RoBoard RB-100 [11], por dicho motivo se decidió escoger una CPU de tamaño mínimo (el sistema de actuación y el sistema de sensorización quedan excluidos, ya

que se encuentran alojados en el interior del "soporte" del robot asistencial ASIBOT).

Otro requisito que debe cumplir la RoBoard RB-100 es disponer de una ranura miniPCI, que es necesaria para la tarjeta de CAN que se ha empleado en esta aplicación. También la BIOS debe permitir arrancar la RoBoard RB-100 desde una memoria flash para ahorrar espacio, ya que se pretende prescindir de lector de CDs, disquetera y disco duro.

Como se verá más adelante, la conexión entre la RoBoard RB-100 y el PC se realizara por medio de Ethernet.

El dispositivo escogido por cumplir todos los requisitos anteriores es la CPU RoBoard RB-100 de la compañía DMP Electronic Inc mostrado en la siguiente figura:

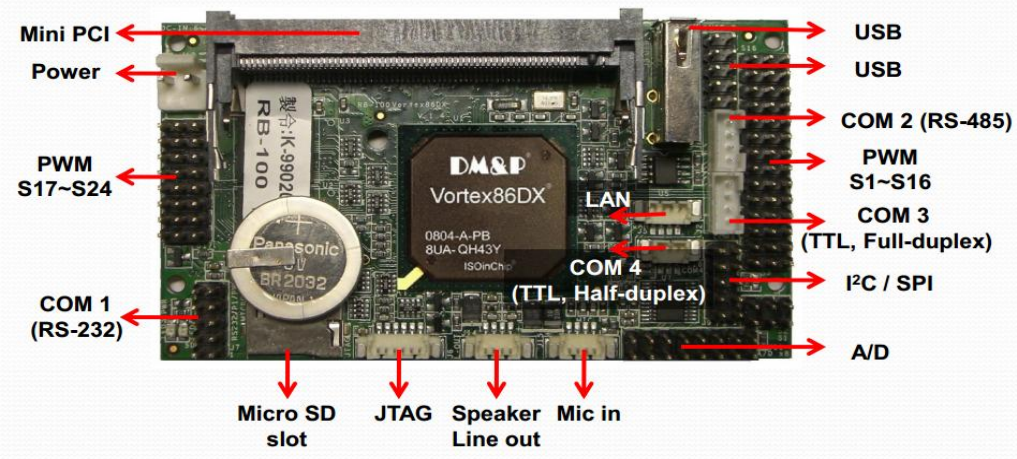


Figura 36: RoBoard RB-100 (imagen de planta).

Figura 37: RoBoard RB-100 de la compañía DMP Electronic Inc.

Características principales

- Potente y pequeño ordenador dedicado para aplicaciones de robótica.
- Procesador Vortex86DX, a 1000 MHz y 256 MB de memoria DDR2.
- Dimensiones 96 mm x 56 mm.
- Peso 40 g.
- Alimentación a DC 6 V a 24 V.
- BIOS: AMI BIOS.
- I/O interfaces: 1 puerto Micro SD; 1 puertos USB V2.0.
- Compatible con Windows, Linux y DOS.
- *Open Source C++ Library* para RoBoard con funciones de I/O (sensores, motores, etc.).
- 2 puertos TTL COM.
- 1 puerto serie RS-232 (Resolución: 115200 bps).
- 1 puerto serie RS-485 (Resolución 115200 bps).
- Bus I^2C (Resolución: 1 Kbps a 3.3 Mbps).
- Bus SPI (Resolución: 10 Mbps a 150 Mbps).

Estructura de microprocesador Vortex86DX de la CPU RoBoard RB-100.

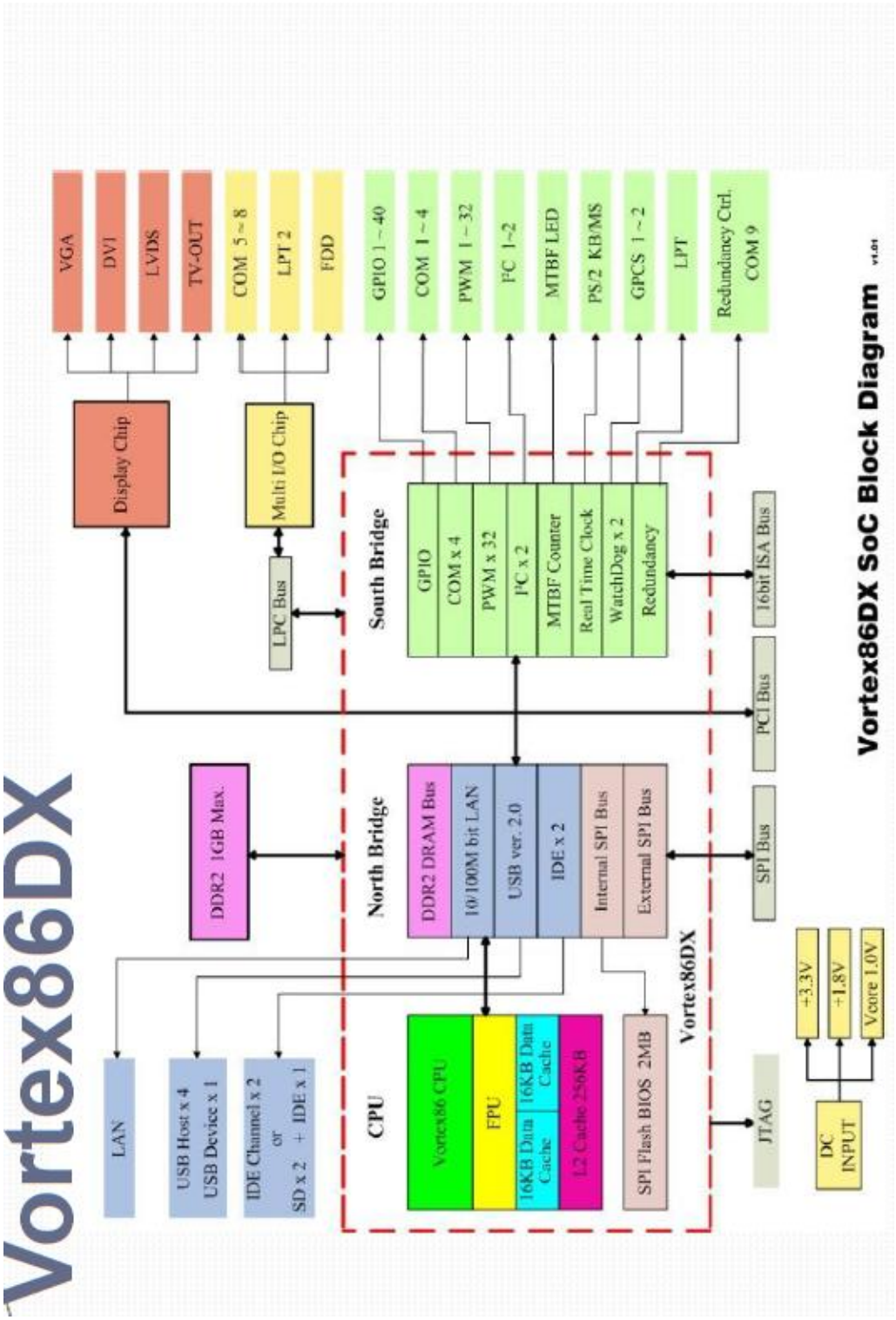


Figura 38: Estructura del Microprocesador Vortex86DX.

Capitulo 4: Desarrollo de la aplicación

4.1 Introducción

Una vez se ha comentado tanto los dispositivos de hardware y herramientas de software empleados, en este capítulo se describe cómo se ha implementado la arquitectura de control desarrollada en este Trabajo de Fin de Grado.

Como se explico en el primer capítulo, la implementación de la arquitectura de control del motor alojado en el interior del soporte del robot asistencial ASIBOT, consta de tres fases claramente diferenciadas:

1. **Fase de montaje del sistema**: en esta fase se muestra todo el conexionado del hardware empleado para desarrollar este Trabajo de Fin de Grado. Se muestran las conexiones de todos los dispositivos conectados al driver ISCM8005, también se explicarán las conexiones tanto de la tarjeta de comunicación HICO.CAN-miniPCI, cómo de la CPU RoBoard RB-100. El soporte del robot asistencial ASIBOT se encuentra montado en la cremallera, por lo que no se mostraran los detalles del montaje.
2. **Fase de configuración del driver que controla los motores:** esta fase se centra en describir los pasos a seguir para configurar el motor que compone dicho sistema. Esta segunda fase se subdivide en tres subfases más. En la primera subfase se configura el motor y elementos cómo el encoder. En la segunda subfase se realiza la configuración de los reguladores que intervienen en el control del motor, es decir, se tiene que ajustar el PID que controla el proceso, también se ajustan parámetros como la velocidad de transmisión, el tipo de control y protecciones frente a sobreintensidades. Y por último, se muestra el desarrollo de la secuencia creada para la inicialización en lenguaje de programación de movimientos TML. También se incluye en esta fase la programación de la memoria EEPROM con el software EEPROM Programmer.
3. **Fase de implementación de las funciones de C/C++:** en esta última fase se muestran la creación de las funciones encargadas del control del motor.

4.2 Fase de montaje del sistema (1º Fase)

4.2.1 Montaje del driver ISCM8005

Todos los pasos que se muestran a continuación vienen explicados en el manual técnico del driver [12].

4.2.1.1 Configuración motor-sensor soportada

Para el desarrollo de la arquitectura de control para gobernar un motor, se ha escogido la siguiente configuración motor-encoder. En la siguiente figura se puede observar dicha configuración:

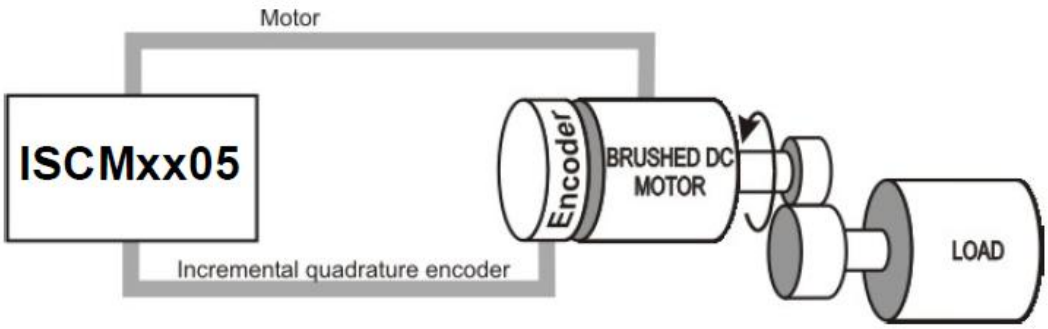


Figura 39: DC *brushed* motor rotatorio con encoder incremental en motor.

Usando está configuración se puede realizar el control de un motor rotatorio de corriente continua, tanto en posición, velocidad o par. El encoder incremental se encuentra en el eje del motor y no en la carga.

Los factores de escala se encargan de realizar la conversión entre unidades del sistema internacional [SI] y las unidades internas [IU] del propio driver. Estos factores de escala deben tener en cuenta la relación de transmisión entre el motor y la carga. Por lo tanto los comandos de posición, velocidad o aceleración que estén expresados en unidades del sistema internacional [SI] se refieren a la carga, mientras que esos mismos comandos, expresados en unidades internas [IU], se refieren al motor.

4.2.1.2 Diseño de conectores

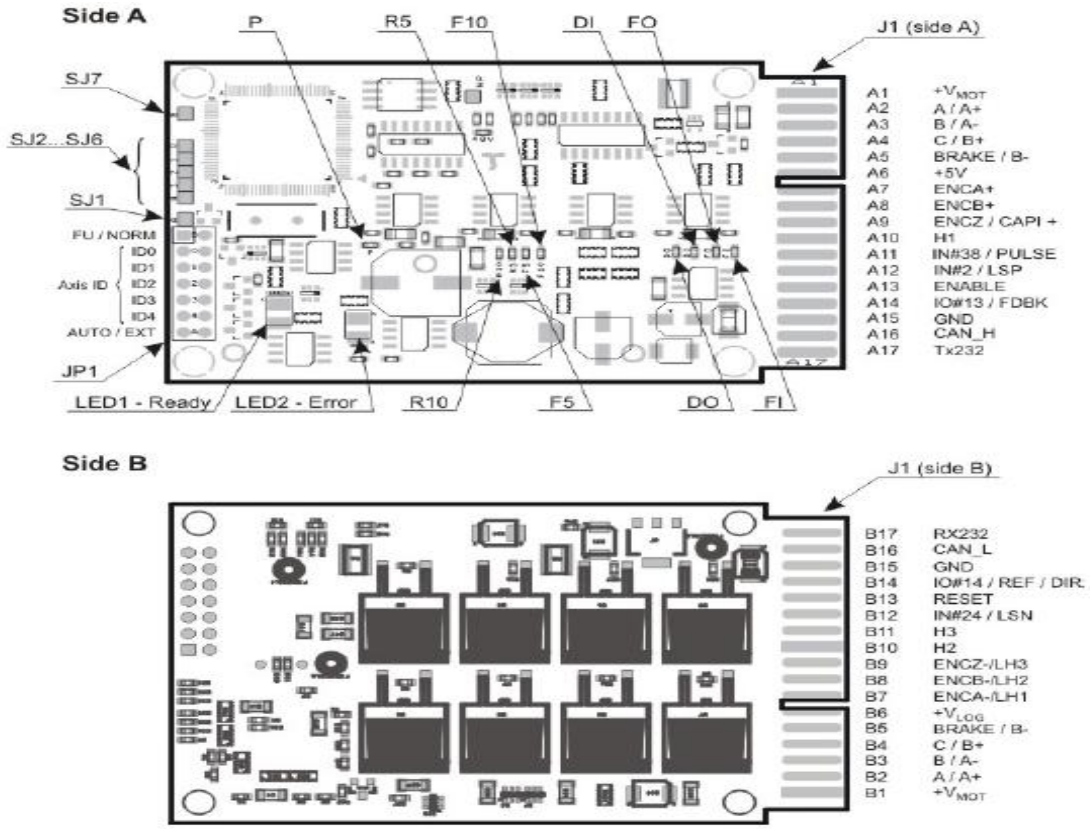


Figura 40: Cara A y Cara B del driver ISCM8005.

Cara A		Cara B	
A1	$+V_{mot}$ (input)	B1	$+V_{mot}$ (input)
A2	A+	B2	A+
A3	A-	B3	A-
A6	+5 V (salida interna)	B6	+ V_{log} (+12 a +48 V_{DC}) (input)
A7	ENC A+	B7	ENC A-
A8	ENC B+	B8	ENC B-
A9	ENC Z+	B9	ENC Z-
A12	LSP (limit switch positive)	B12	LSN(limit switch negative)
A15	GND	B15	GND
A16	CAN_H (línea positiva)	B16	CAN_L (línea negativa)
A17	TX232	B17	RX323

Tabla 31: Descripción de los conectores empleados del driver ISCM8005.

4.2.1.3 Elección del ID (identificador) del eje en el driver ISCM8005

Para definir el número de eje con el que vamos a identificar el driver ISCM8005, es necesario realizar una soldadura en la cara A del driver. La localización de la soldadura es la que se muestra a continuación:

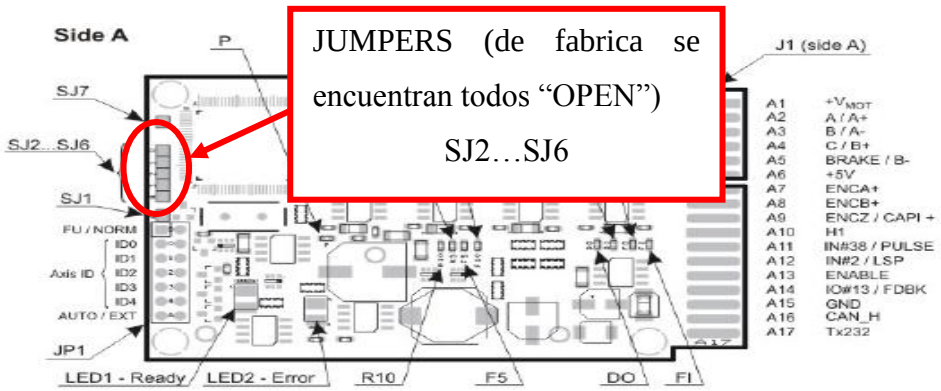


Figura 41: JUMPERS soldables utilizados para identificar el numero de eje del driver.

Ahora que están localizados los JUMPERS, es necesario saber el modo en que los vamos a soldar, para esto es necesario acudir a la siguiente tabla:

JP1 position					AXIS ID
11-12	9-10	7-8	5-6	3-4	
ID – Bit4	ID – Bit3	ID – Bit2	ID – Bit1	ID – Bit0	
OPEN	OPEN	OPEN	OPEN	OPEN	255
OPEN	OPEN	OPEN	OPEN	SHORT	1
OPEN	OPEN	OPEN	SHORT	OPEN	2
OPEN	OPEN	OPEN	SHORT	SHORT	3
OPEN	OPEN	SHORT	OPEN	OPEN	4
OPEN	OPEN	SHORT	OPEN	SHORT	5
OPEN	OPEN	SHORT	SHORT	OPEN	6
OPEN	OPEN	SHORT	SHORT	SHORT	7
OPEN	SHORT	OPEN	OPEN	OPEN	8
OPEN	SHORT	OPEN	OPEN	SHORT	9
OPEN	SHORT	OPEN	SHORT	OPEN	10
OPEN	SHORT	OPEN	SHORT	SHORT	11
OPEN	SHORT	SHORT	OPEN	OPEN	12
OPEN	SHORT	SHORT	OPEN	SHORT	13
OPEN	SHORT	SHORT	SHORT	OPEN	14
OPEN	SHORT	SHORT	SHORT	SHORT	15
SHORT	OPEN	OPEN	OPEN	OPEN	16
SHORT	OPEN	OPEN	OPEN	SHORT	17
SHORT	OPEN	OPEN	SHORT	OPEN	18
SHORT	OPEN	OPEN	SHORT	SHORT	19
SHORT	OPEN	SHORT	OPEN	OPEN	20
SHORT	OPEN	SHORT	OPEN	SHORT	21
SHORT	OPEN	SHORT	SHORT	OPEN	22

Tabla 32: Selección del ID de eje por medio la soldadura de JUMPERS.

Para establecer el *axis ID* (identificador de eje), hemos acudido a la tabla anterior, como queremos el identificador numero '1', tenemos que configurar los pines como se muestra en dicha tabla. Únicamente hay que soldar el pin 3-4, quedando el resto como vienen de fábrica.

Una vez esta determinado que posición es necesaria soldar, acudimos a la siguiente figura, para determinar que unión hay que soldar.

Jumper	Solder joint
JP1 position 1-2	SJ7
JP1 position 3-4	SJ6
JP1 position 5-6	SJ5
JP1 position 7-8	SJ4
JP1 position 9-10	SJ3
JP1 position 11-12	SJ2
JP1 position 13-14	SJ1

Figura 42: Tabla de soldaduras.

A continuación se muestra una imagen del driver con la soldadura realizada, para facilitar la comprensión de lo explicado en este apartado.

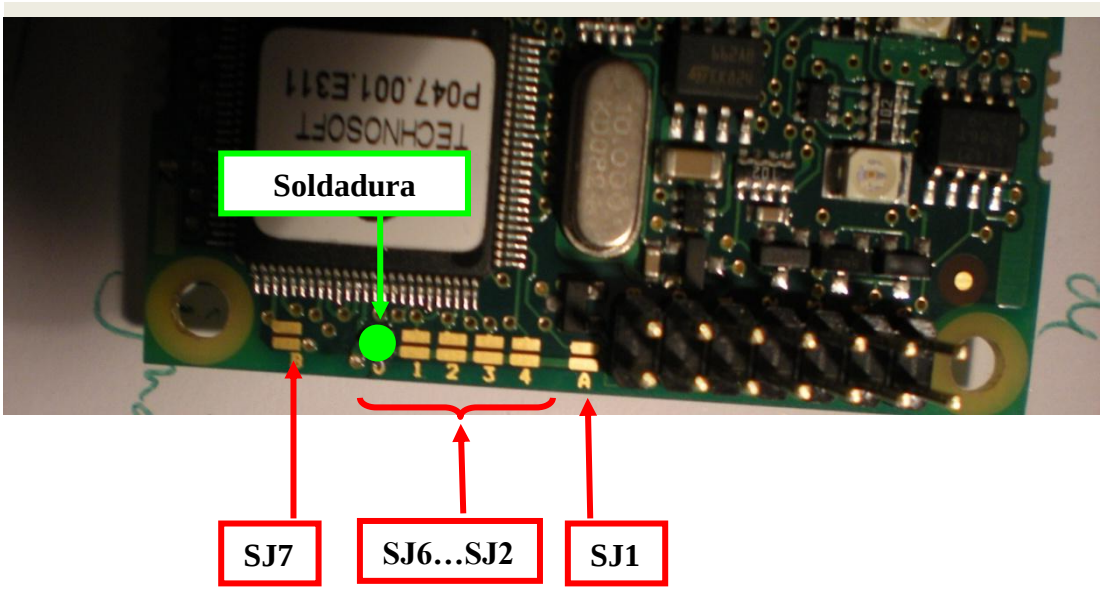


Figura 43: Localización de la soldadura.

4.2.1.4 Conexión de la alimentación del driver ISCM8005

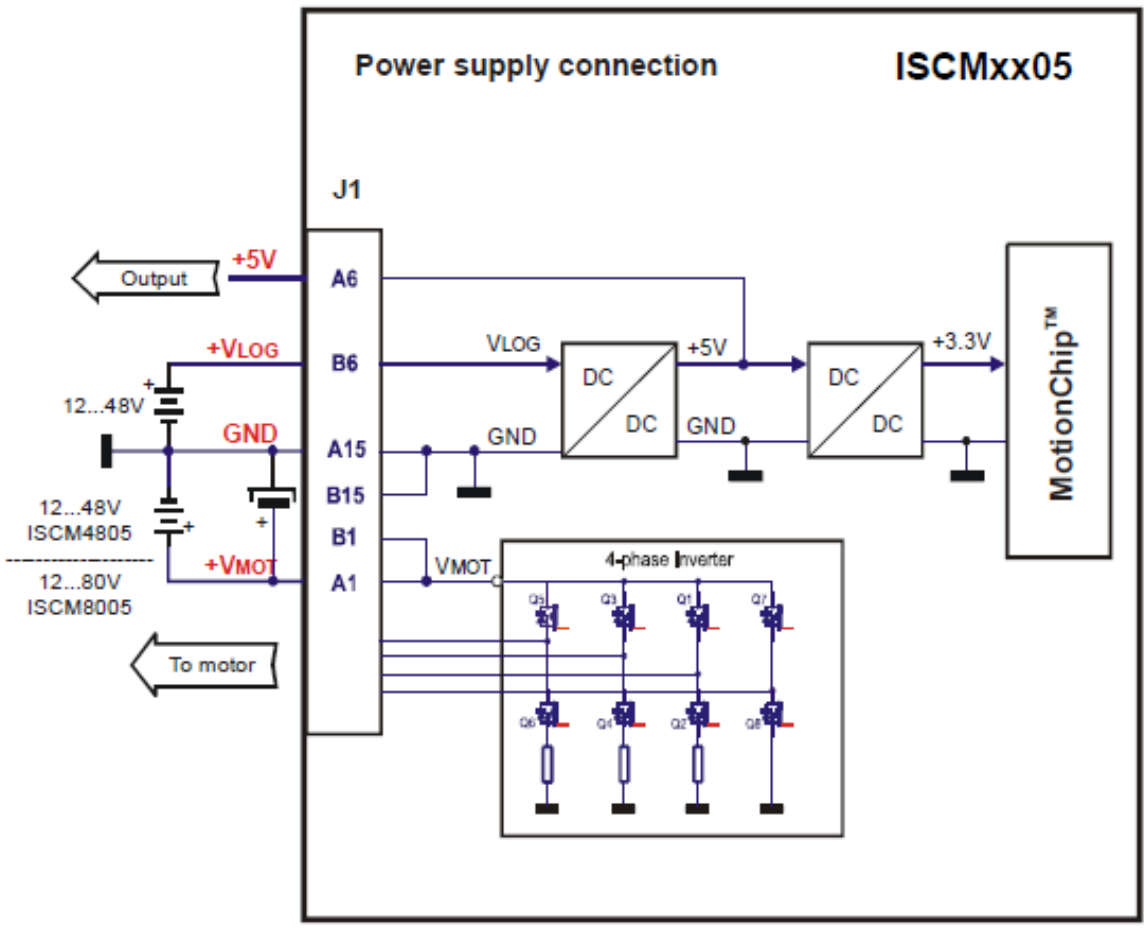


Figura 44: Conexionado de la alimentación del driver ISCM8005.

Este dispositivo requiere una fuente de alimentación de entre 12 V y 80 V ($+V_{mot}$) para la potencia y otra fuente de entre 12 V y 48 V ($+V_{log}$). El conexionado realizado consiste en conectar ambas tomas ($+V_{mot}$ y $+V_{log}$) a las misma fuente de 24 V para simplificar el circuito final.

También se deberá colocar un condensador a la entrada de la alimentación de valor igual o superior a 100µF.

4.2.1.5 Conexión del motor al driver ISCM8005

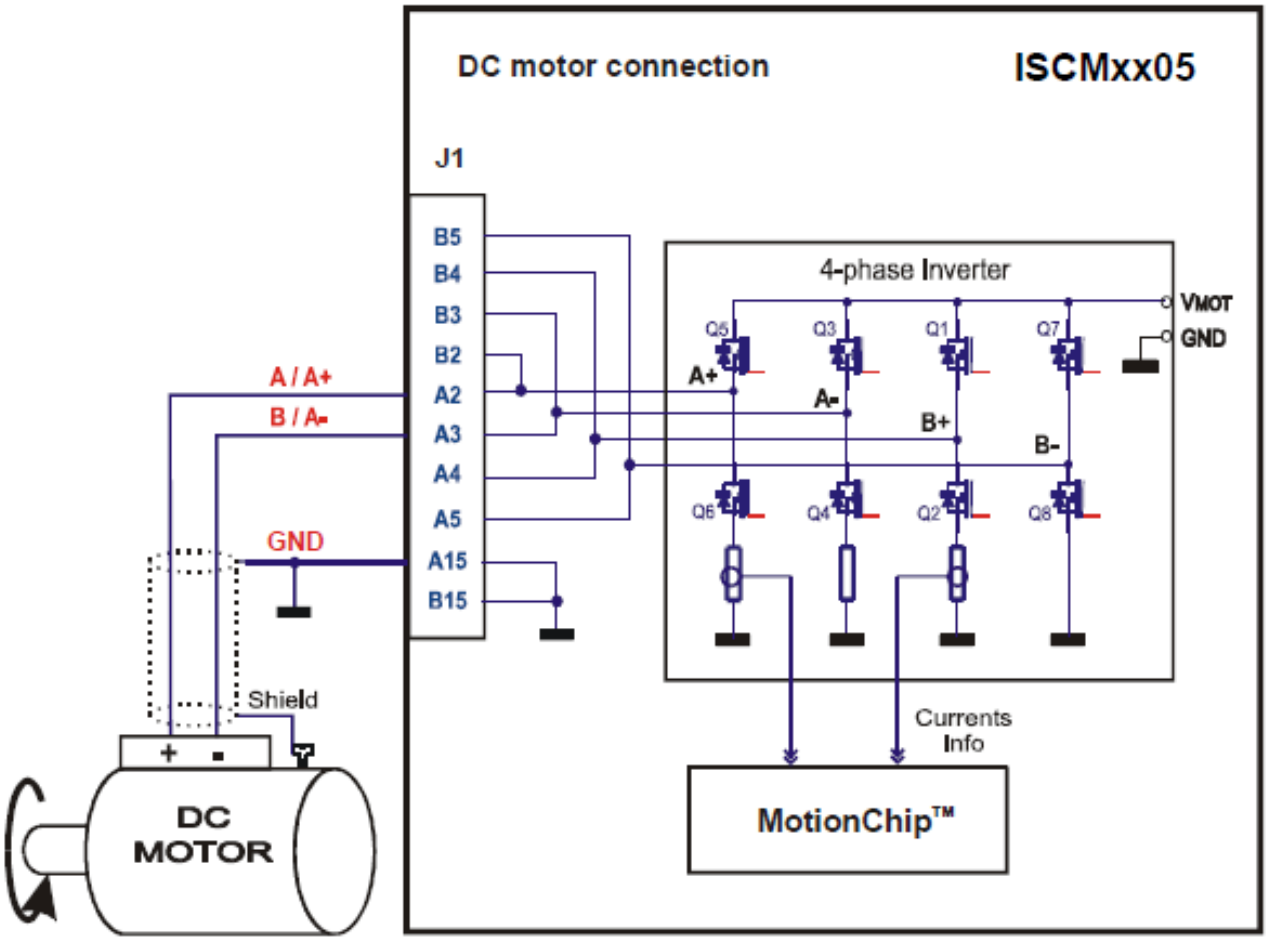


Figura 45: Conexión del motor al driver ISCM8005.

El dispositivo actuador es un motor de corriente continua, por lo que su sus conexiones con el driver corresponden a un terminal positivo y otro negativo.

4.2.1.6 Conexión del encoder relativo diferencial al driver ISCM8005

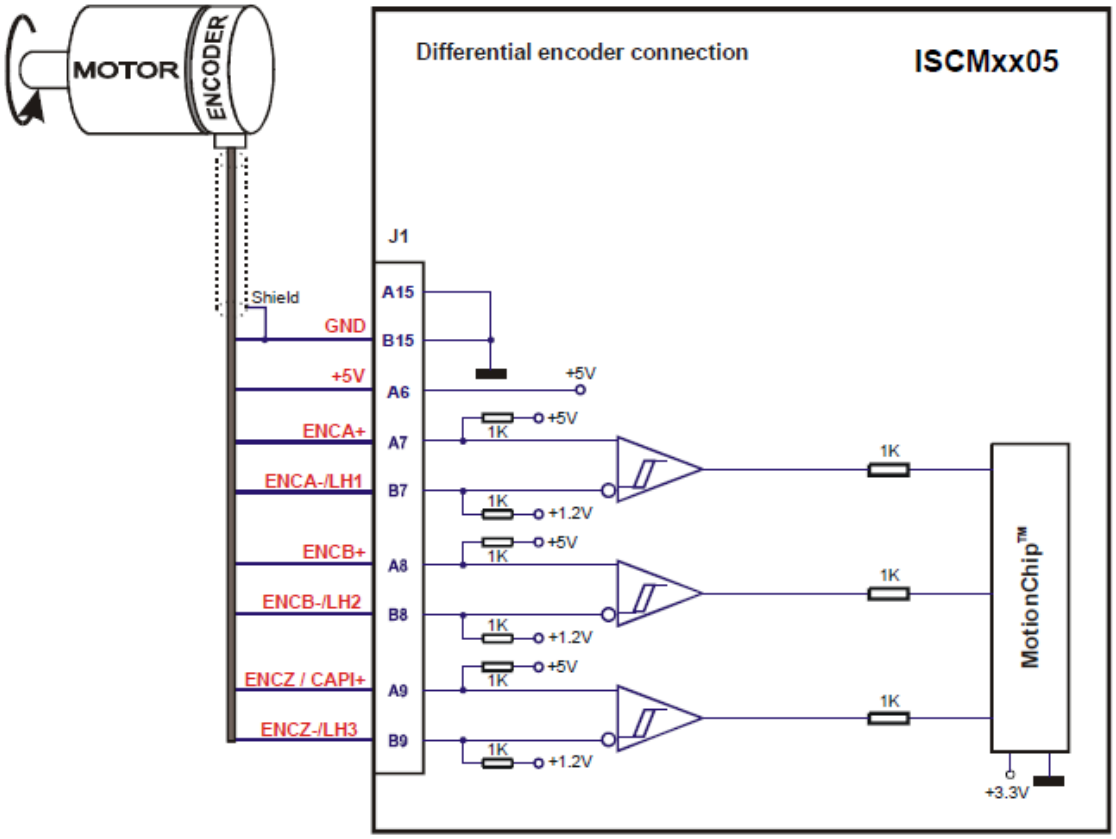


Figura 46: Conexión del encoder relativo diferencial al driver ISCM8005.

El dispositivo sensor utilizado es un encoder relativo diferencial, que es alimentado con 5 V. Utiliza 3 señales de comunicación: 2 para determinar el sentido de giro y otro como índice. Al ser diferencial, estas 3 señales son enviadas tanto negadas como sin negar, para evitar fenómenos de reflexión.

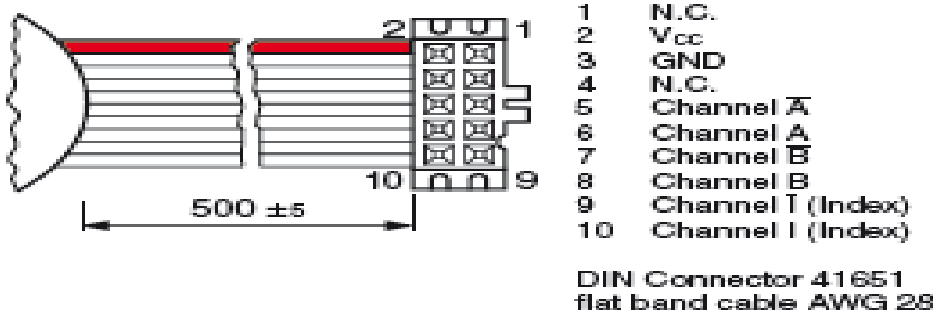


Figura 47: Asignación de pines del encoder.

4.2.1.7 Conexión de la comunicación CAN

R = 120 Ω

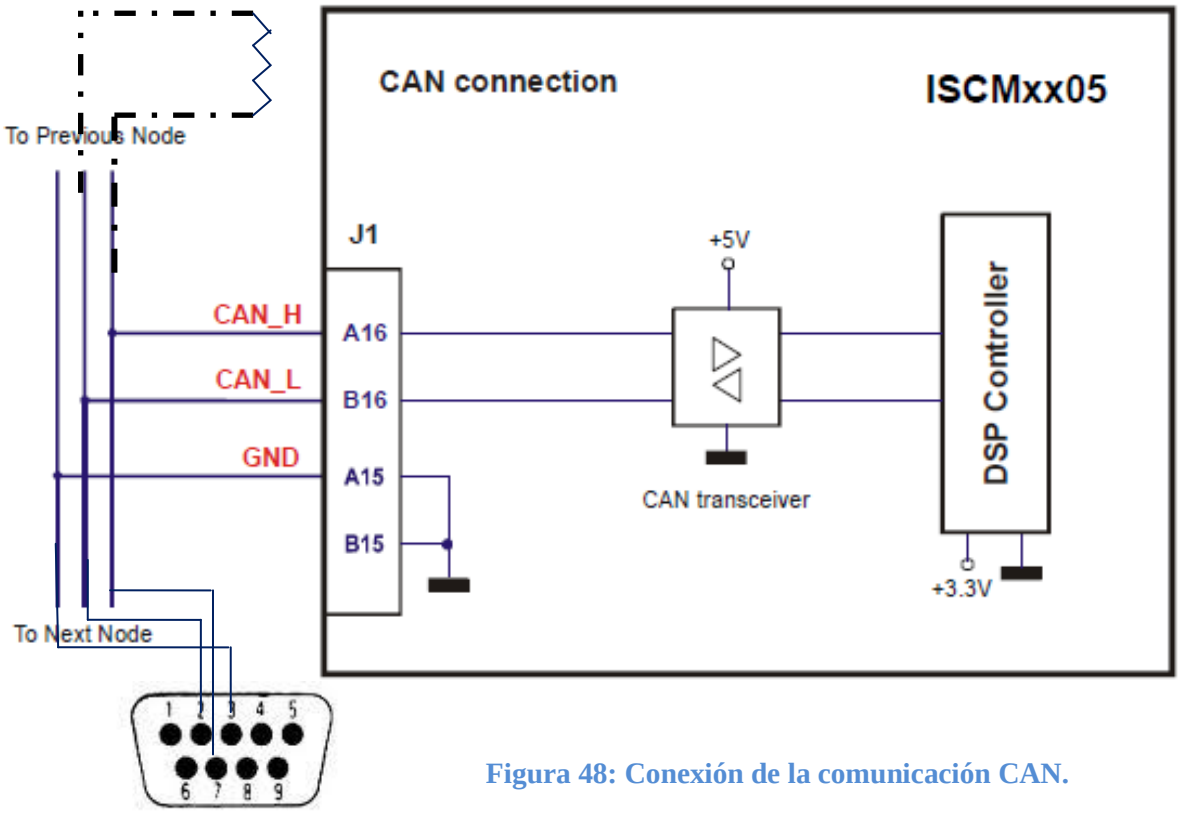


Figura 48: Conexión de la comunicación CAN.

Para que el driver pueda pertenecer al bus de campo CAN, es necesario realizar el conexionado de los pines del driver CAN_H, CAN_L y GND a un conector DSUB-9 de la misma manera que lo implementa la tarjeta CAN (HICO.CAN-miniPCI).

En los extremos del bus CAN deben ir colocadas dos resistencias de 120 Ω, para evitar posibles reflexiones. Solo una por extremo.

4.2.1.8 Conexión de la comunicación RS-232

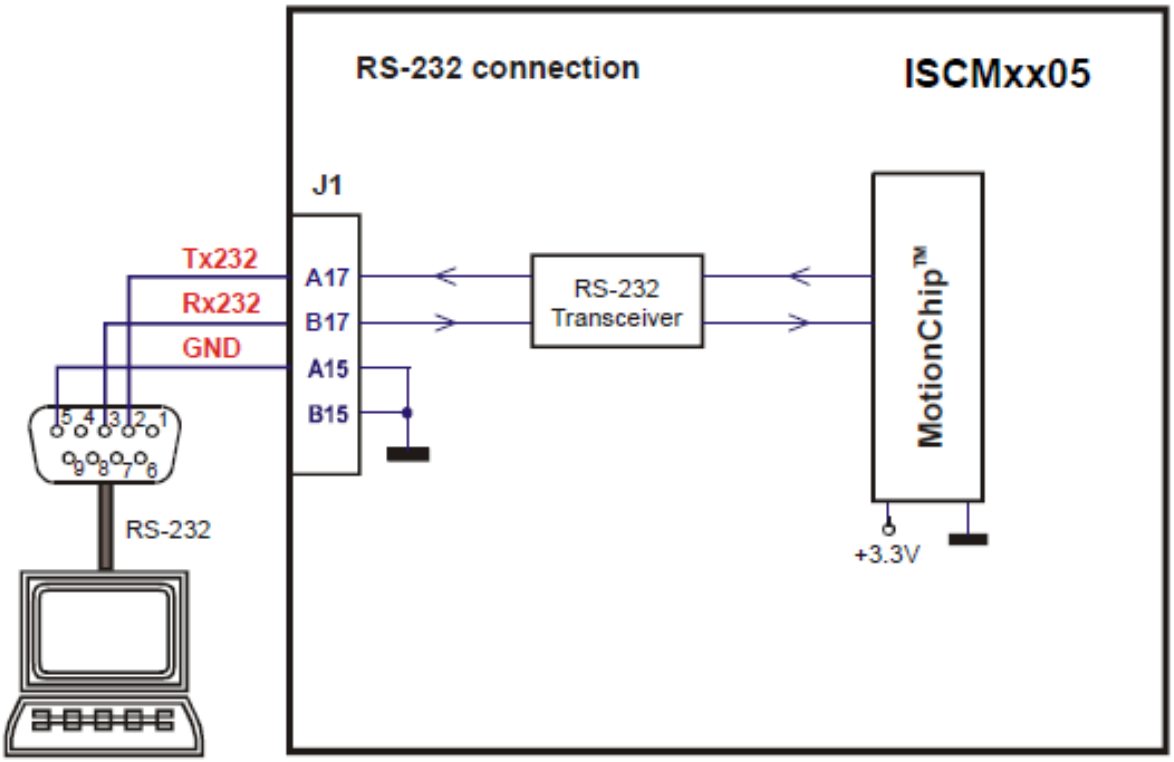


Figura 49: Conexión de la comunicación RS-232.

Este conexionado se realiza únicamente para realizar la configuración inicial del driver mediante el programa EasySetup o EasyMotion Studio proporcionado por Technosoft. Para conectar el driver con el puerto serie de un PC con software instalado, se ha empleado un conector DSUB-9.

Aparte se ha utilizado un adaptador puerto serie-USB, para poder conectar el puerto de comunicación serie RS-232 con un PC con puertos USB.

4.2.1.9 Conexión de los finales de carrera al driver ISCM8005

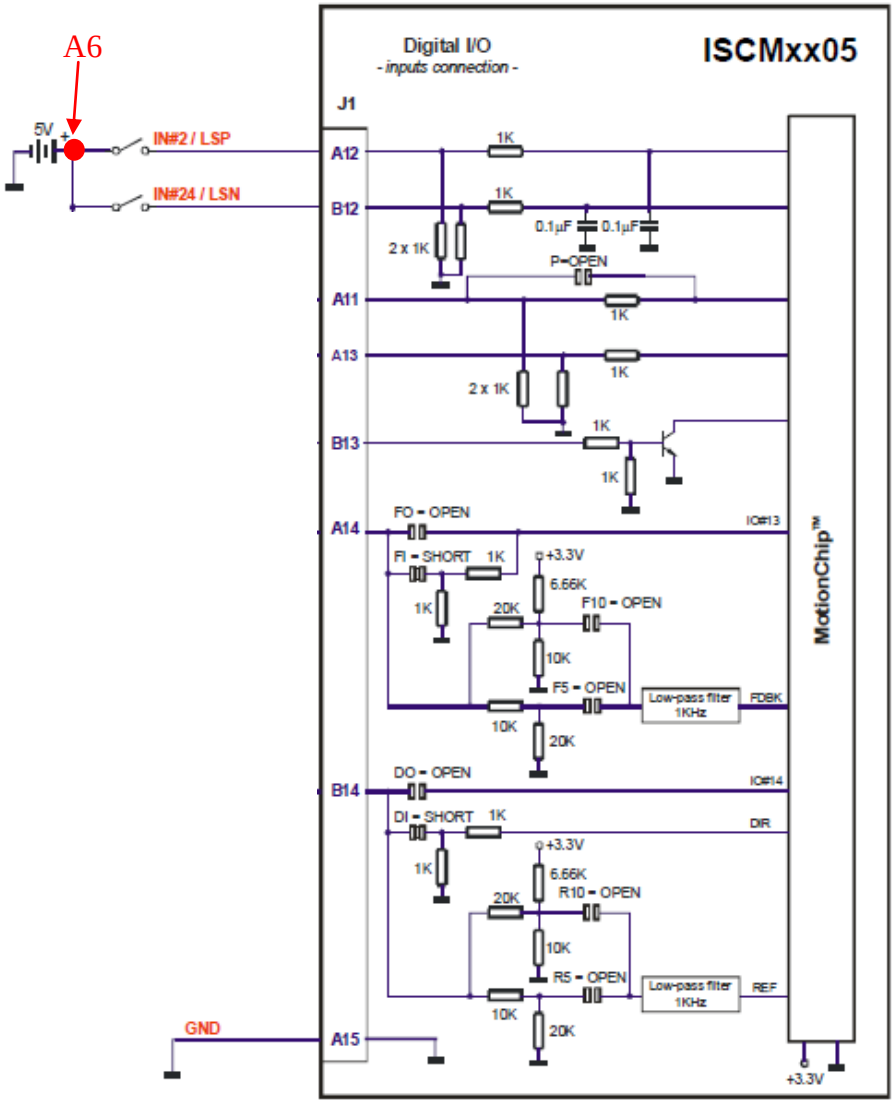


Figura 50: Conexión de los finales de carrera al driver ISCM8005.

El montaje de los finales de carrera es como se indica en la figura anterior. Un extremo del sensor final de carrera se conecta al pin A6 del driver, que es una alimentación interna de la placa a 5 V, y el otro extremo se conecta al pin A12 o B12 según el límite que le corresponda a dicho final de carrera. Ambos finales de carrera son normalmente cerrados, cuando son pulsados el circuito quedaría abierto, mientras no sean pulsados, el circuito permanece cerrado.

4.2.1.10 Integración de todos los dispositivos en la placa de pruebas

Todos los dispositivos conectados anteriormente se integran en una placa de pruebas, para poder hacer más sencillo el montaje del driver ISCM8005 con todos los demás elementos. A continuación se muestra una imagen de la placa de pruebas utilizada.

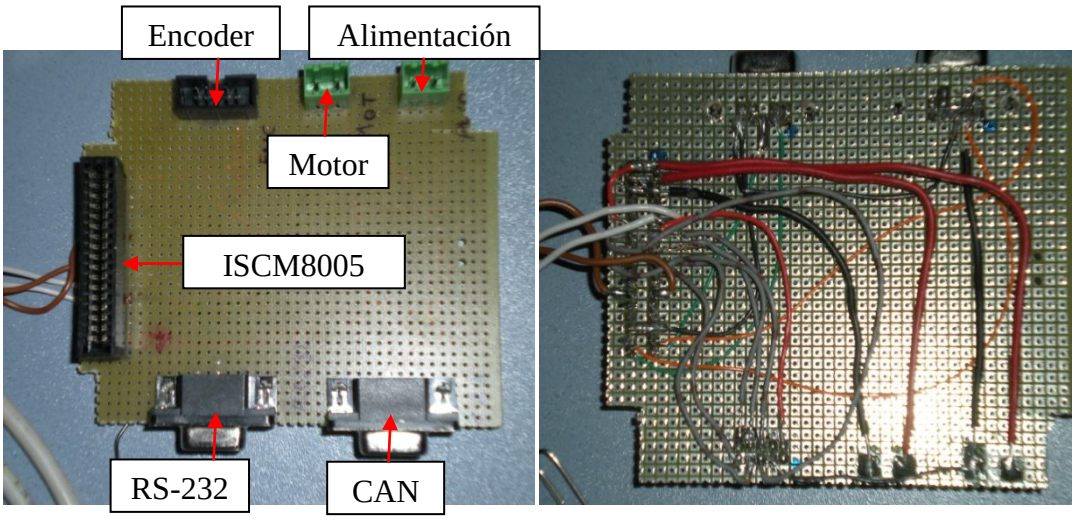


Figura 51: Placa de pruebas.

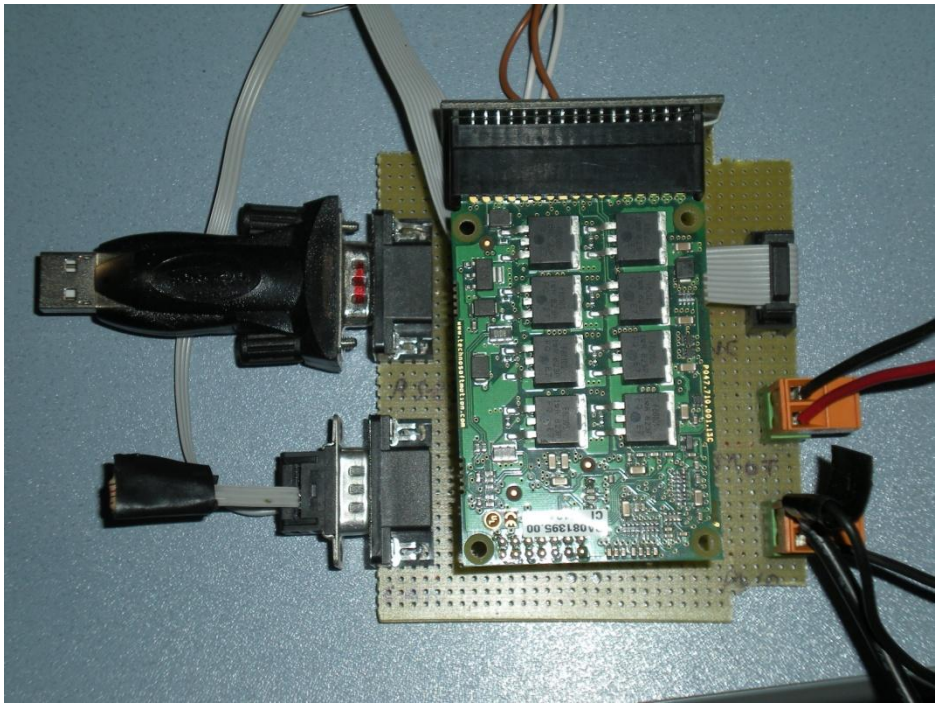


Figura 52: Placa de pruebas con todos los dispositivos conectados.

4.2.2 Montaje de la tarjeta de comunicación HICO.CAN-miniPCI

Para poder llevar a cabo el montaje de la tarjeta de comunicación HICO.CAN-miniPCI, es necesario hacer dos acciones.

La primera acción es comprobar que el puerto de miniPCI con la que cuenta la CPU RoBoard RB-100 está disponible.

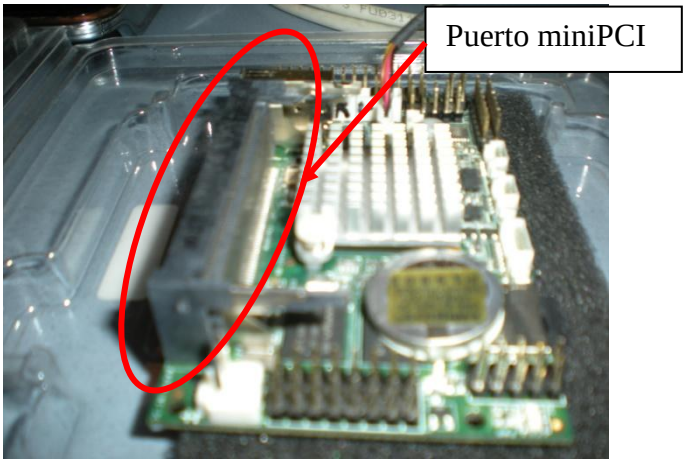


Figura 53: Puerto miniPCI disponible (en imagen RoBoard RB-100).

En caso contrario, retirar la tarjeta que se encuentre conectada. Una vez que el puerto miniPCI está disponible conectar la tarjeta de comunicaciones en dicho puerto, quedando como se muestra en la siguiente figura.

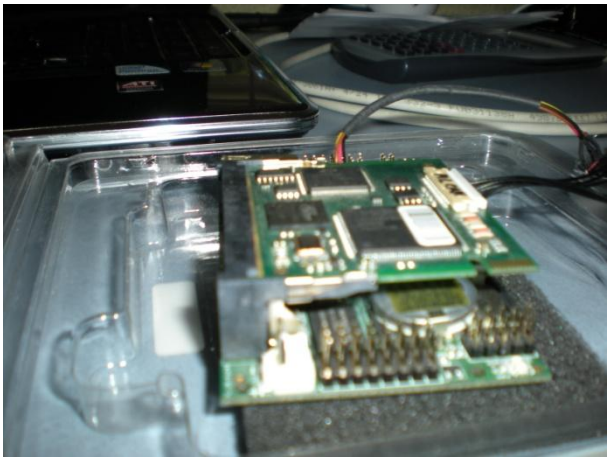


Figura 54: Tarjeta de comunicaciones HICO.CAN-miniPCI conectada a la CPU RoBoard RB-100.

La segunda acción a realizar una vez se ha completado el paso anterior, es conectar el puerto CAN1 (la tarjeta HICO.CAN miniPCI dispone de dos canales de comunicación CAN1 y CAN2) con el puerto CAN disponible en el driver ISCM8005.

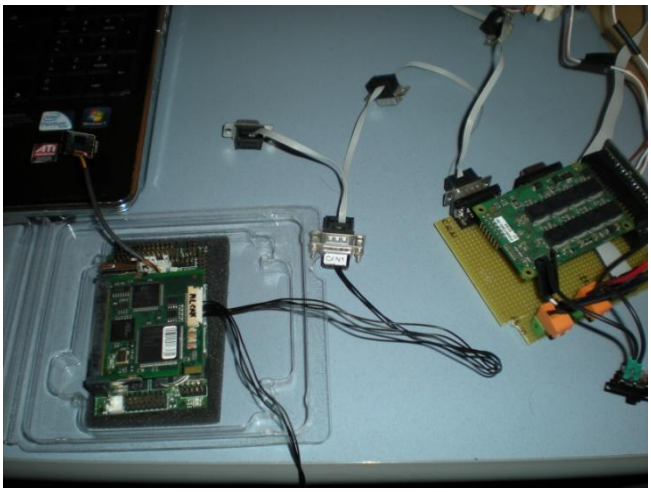


Figura 55: Conexionado de la tarjeta HICO.CAN-miniPCI con el driver ISCM8005.

La conexión se realiza por medio de un bus CAN de datos, con tres hilos, una resistencia de 120 Ω en cada extremo del bus y conectores DSUB-9 (todos los conectores son "machos", excepto el primero que es "hembra" para poder conectar el bus con el puerto CAN1 de la tarjeta de comunicaciones HICO.CAN-miniPCI).

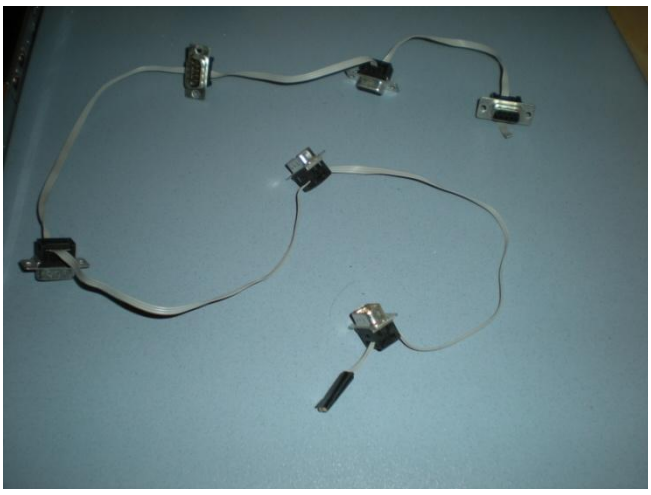
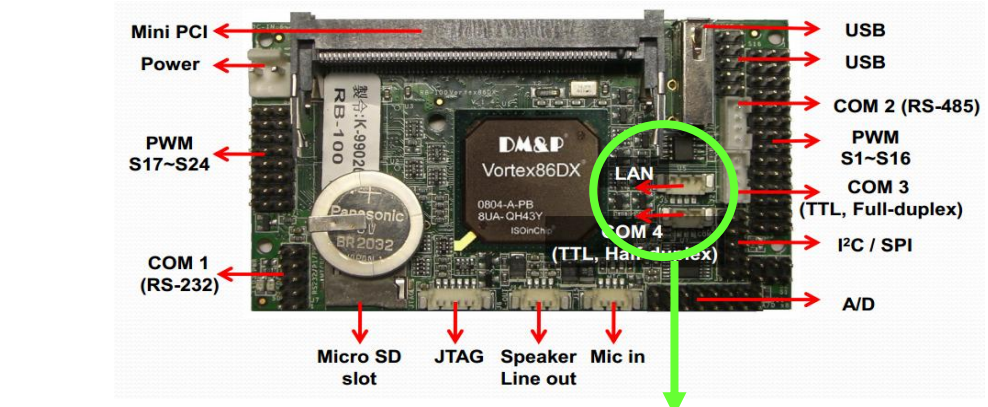


Figura 56: Cable empleado para la comunicación CAN.

4.2.3 Montaje de la CPU RoBoard RB-100

Como se ha visto en el apartado anterior, la tarjeta de comunicación HICO.CAN-miniPCI va conectada en el puerto miniPCI de la CPU RoBoard RB-100, por lo tanto no hay que volver a realizar este paso.

Para el montaje de la CPU RoBoard RB-100 únicamente es necesario realizar dos pasos: el primer paso es establecer la comunicación Ethernet entre la CPU RoBoard RB-100 y el PC empleado para editar y probar los programas implementados en C/C++. Para empezar hay que utilizar el cable LAN disponible en el paquete de cables que proporciona la propia marca. Una vez que se tiene el cable, se conecta en los pines del 1 al 4, como se muestra a continuación.



Pin #	Signal Name	Pin #	Signal Name
1	LAN-TX+	2	LAN-TX-
3	LAN-RX+	4	LAN-RX-

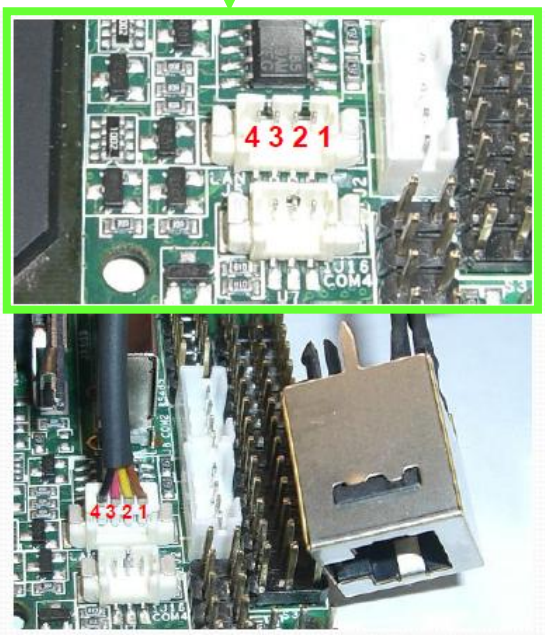


Figura 57: Conexionado del cable LAN a la CPU RoBoard RB-100.

Una vez se ha montado el cable LAN en la placa Roboard RB-100, empleando un cable cruzado de red como el que se muestra en la figura siguiente, se procede a conectar el PC y la placa RoBoard RB-100.

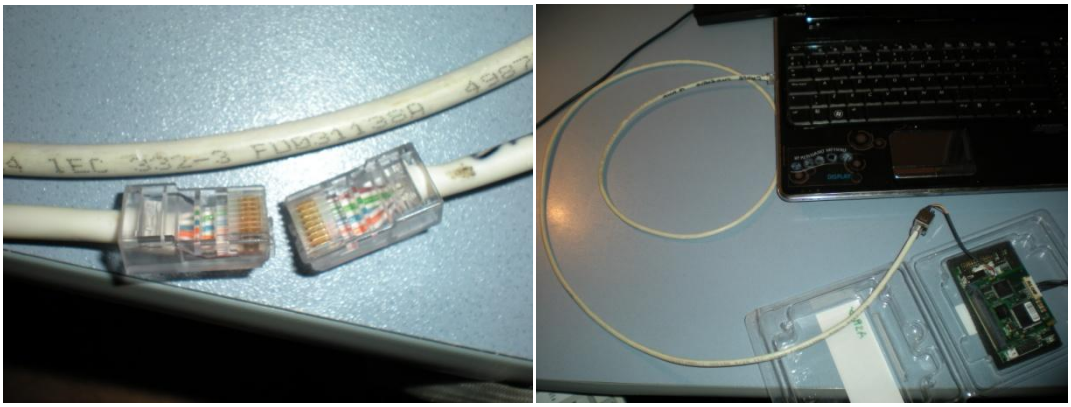
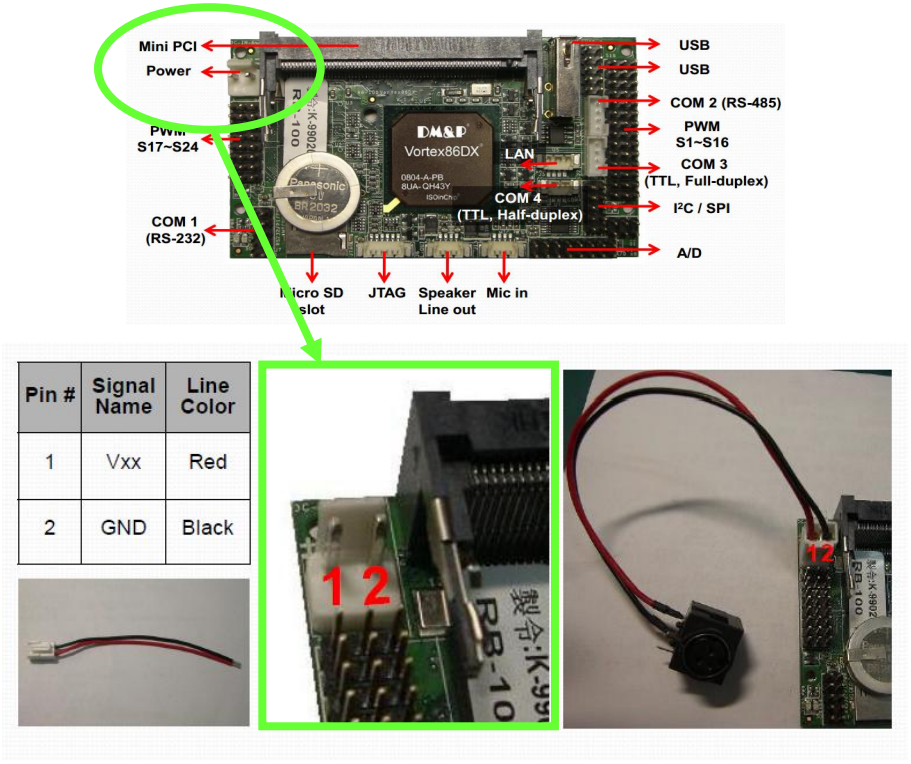


Figura 58: Conexión entre CPU RoBoard RB-100 y PC (Ethernet).

El segundo paso, es conectar la alimentación de la placa a una tensión continua (DC) que puede variar de 6 V a 24 V. Para realizar este paso, se requiere el uso del cable "*Power connector"* que es proporcionado por el fabricante, como ocurre con el cable LAN empleado en el paso anterior. La localización y montaje se muestra en la figura siguiente:



Pin #	Signal Name	Line Color
1	Vxx	Red
2	GND	Black

Figura 59: Conexión de la alimentación en la CPU RoBoard RB-100.

4.3 Fase de configuración del driver que controla los motores (2º Fase)

Para la configuración del driver ISCM8005 se requiere el empleo de las herramientas de software EasyMotion Studio [13] (para crear la secuencia de movimientos que se grabará en la memoria EEPROM del driver ISCM8005) y EasySetup (para la configuración del driver).

Cabe destacar que para el desarrollo de este Trabajo de Fin de Grado se ha utilizado el software EasyMotion Studio únicamente, debido a que se puede crear la secuencia de movimiento en lenguaje TML y configurar el driver sin necesidad de utilizar EasySetup.

En esta fase también se utiliza el software EEPROM Programmer para grabar los datos en la memoria, tanto de configuraciones como secuencias de movimientos.

Una vez se han aclarado los programas que intervienen en esta fase, se inicia el software EasyMotion Studio de Technosoft. Para comenzar se debe escoger un proyecto nuevo. Cuando se pincha en el proyecto nuevo aparece la ventana que se muestra señalada con la flecha roja.

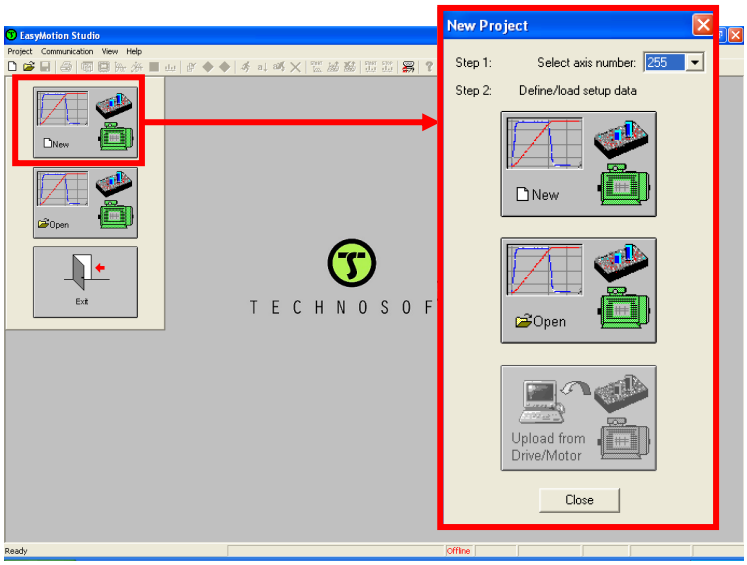


Figura 60: Proyecto nuevo EasyMotion Studio.

Una vez hecho esto, se debe definir el número de eje, para esta aplicación se ha asignado el numero de eje ‘1’. Después se debe escoger el tipo de driver que se está utilizando para desarrollar la aplicación. Los pasos que se deben seguir son los siguientes:

“**new/PlugInDrives/ISCM8005CANopen/BrushedMotor/Incremental_Encoder**”.

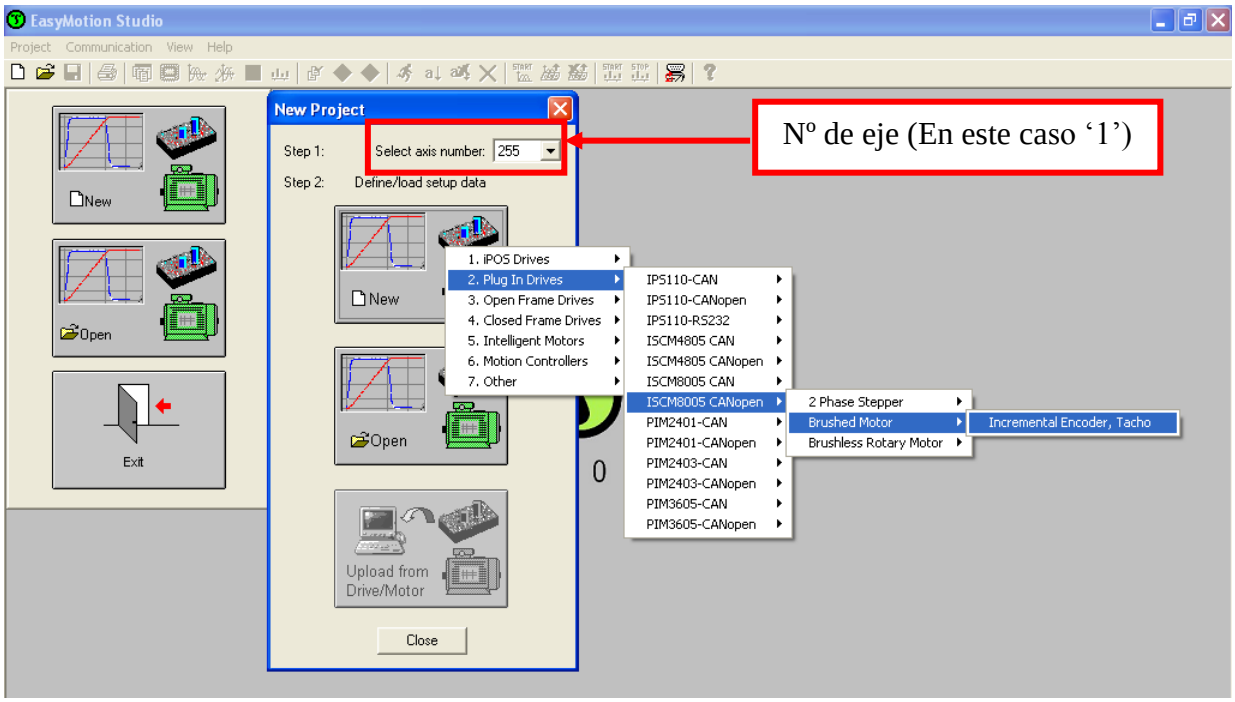


Figura 61: Asignar el número de eje de la placa y escoger el tipo de driver empleado.

Una vez se han realizado los dos pasos mostrados en la anterior figura, la siguiente acción a realizar, es establecer la comunicación entre el driver ISCM8005 y el PC con el que vamos a configurar el driver. Para la comunicación se utiliza el puerto RS-232, que se encuentra en el driver ISCM8005. Por medio de un adaptador USB a RS-232 de un puerto DSUB-9 macho, que se conecta en la placa de pruebas y el USB al PC. Para establecer la comunicación vía RS-232 es necesario realizar los siguientes pasos: **“Communication/Setup” (**aparecerá la figura 60, que se muestra a continuación).

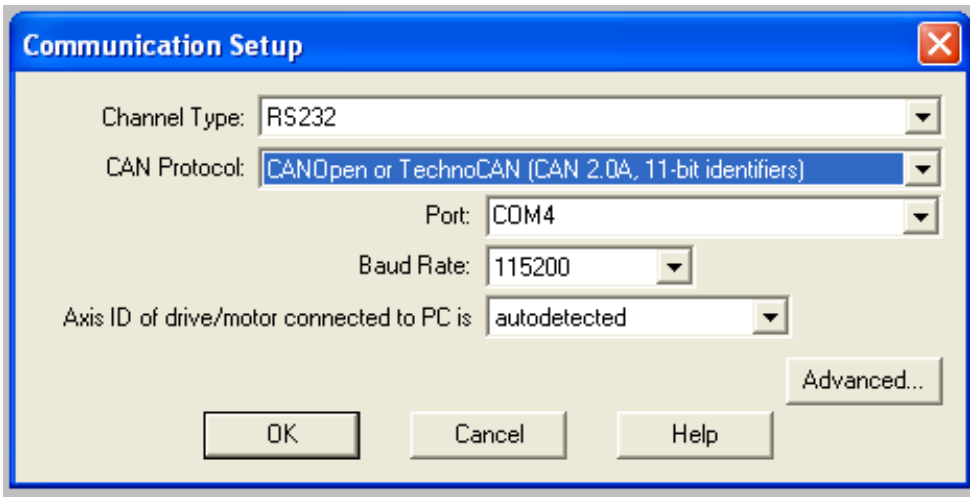


Figura 62: Communication Setup.

Los parámetros que hay que ajustar para la comunicación son los siguientes:

- El canal de comunicación es RS-232.
- El protocolo CAN empleado CANopen.
- Puerto empleado en el PC: COM4.
- Velocidad de transmisión: 115200 baudios.
- Axis ID: Auto detección.

Una vez se ha establecido la comunicación, en la pantalla principal del programa sale un mensaje que pone "online" en verde, si la comunicación no se ha establecido correctamente aparece un mensaje que pone "offline" en rojo.

Cuando la comunicación se encuentra en modo "online", ya podemos realizar la configuración del motor y la configuración del regulador.

4.3.1 Configuración: DC Motor Setup

Cuando el proyecto está creado con el tipo de driver utilizado, se ha establecido la comunicación, el siguiente paso es establecer la configuración del motor que controla el driver.

Este paso consiste en ajustar los valores de control del driver sobre el motor, como puede ser el ajuste de la intensidad de pico al arrancar el motor, el número de líneas del encoder, etc.

La figura que se muestra a continuación, detalla las características del motor que son requeridas en el programa, para poder ajustar el driver al motor que controla.

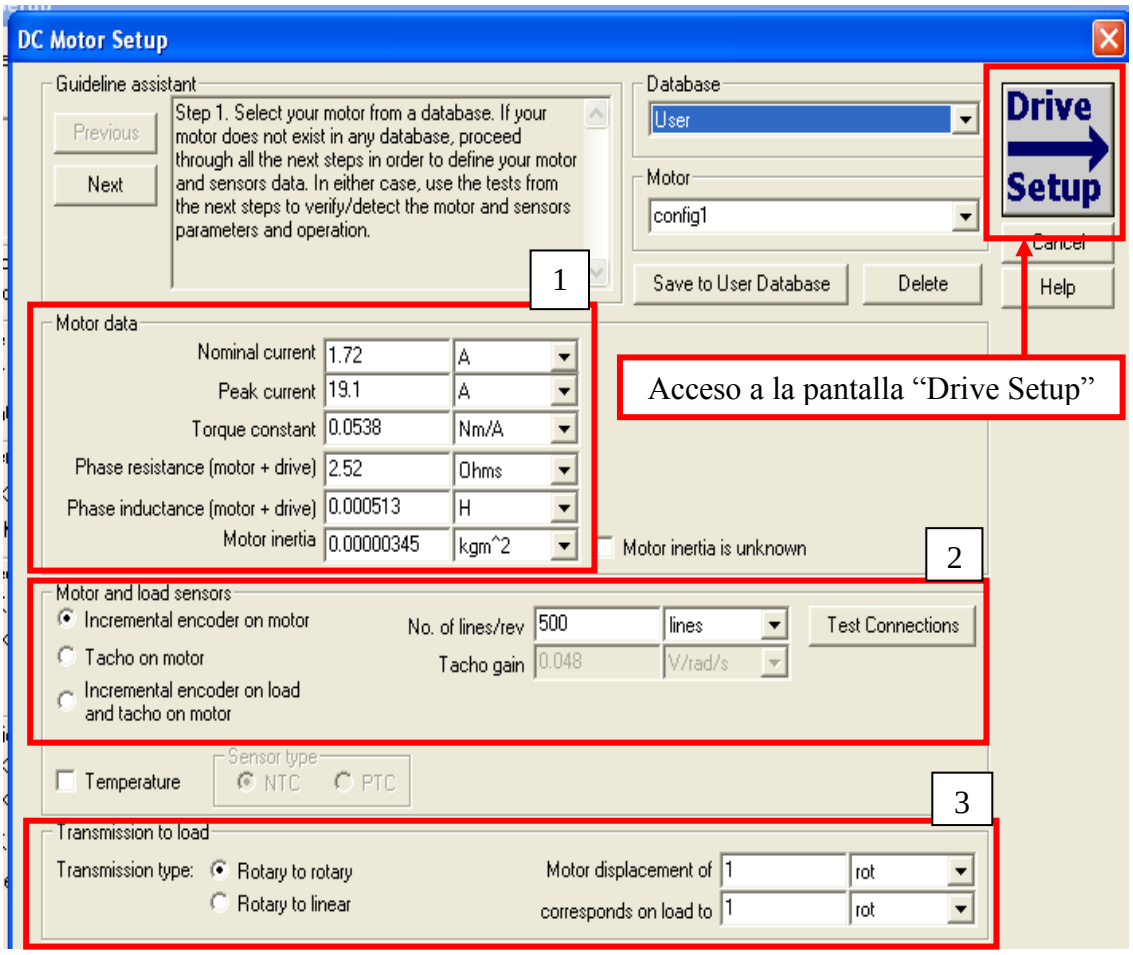


Figura 63: DC Motor Setup.

Los parámetros que hay que ajustar en el "Motor Setup" son los siguientes:

1. **Datos del motor.**
    - Corriente nominal: 1,72 A.
    - Corriente de pico (corriente de arranque): 19,1 A.
    - Constante de par: 0,0538 Nm/A.
    - Resistencia de fase: 2,52 Ω.
    - Inductancia de fase: 0,000513 H.
    - Inercia del motor: 0,00000345 $Kg \cdot m^2$.
2. **Sensor en motor o carga.**
    - Encoder incremental en motor.
    - Número de líneas del encoder: 500 líneas.
3. **Transmisión.**
    - Tipo de transmisión: rotacional a rotacional.
    - Relación de transmisión: 1:104.

Para entender mejor los parámetros que se han introducido, es recomendable ver la Tabla 26 mostrada anteriormente.

4.3.2 Configuración: Drive Setup

Una vez realizada la configuración del motor, procedemos a configurar los reguladores del driver, el modo de control y protecciones frente a excesos de corriente. Esto se realiza en la pantalla de "Drive Setup", a esta pantalla se accede pulsando el botón que aparece marcado en la Figura 61.

La figura que se muestra a continuación, detalla los parámetros necesarios para ajustar los reguladores, protecciones hardware y el modo de control.

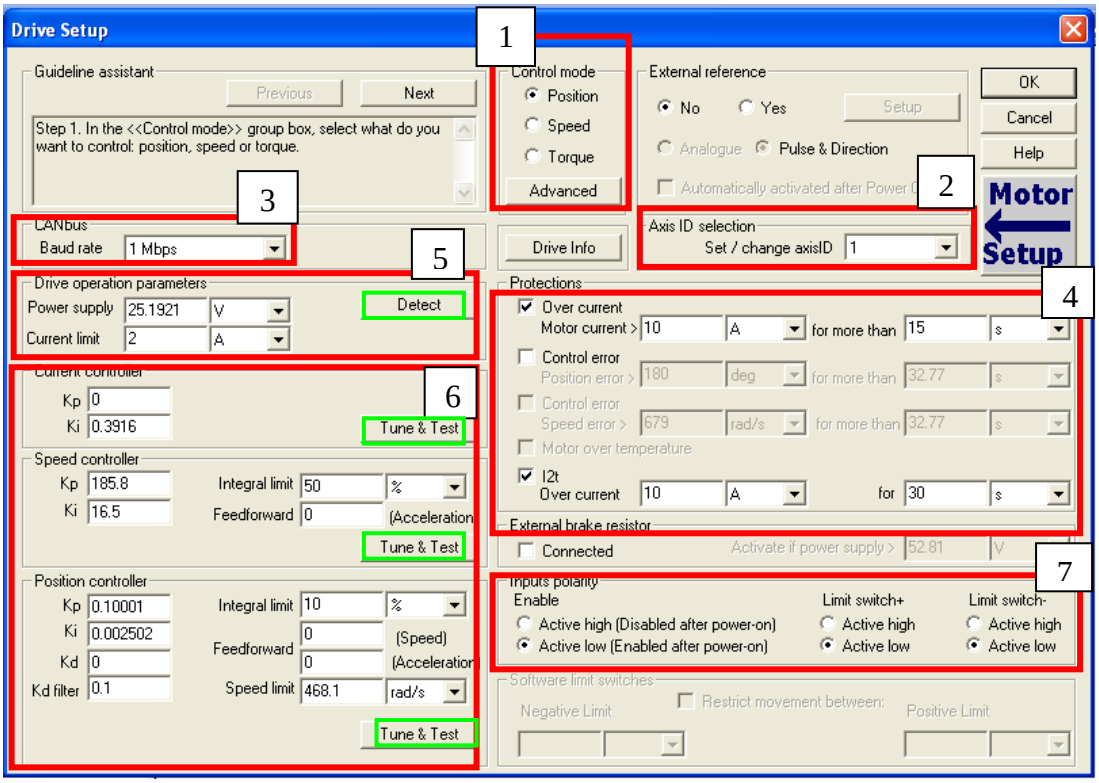


Figura 64: Drive Setup.

A continuación se expondrán los valores empleados para cada parámetro utilizado en la configuración de esta aplicación:

1. **Modo de control.**
   - Control en posición (se puede utilizar otro tipo de control como velocidad o par).
2. **Selección del ID (identificador) del eje.**
   - Ajustar/cambiar Identificado de eje: 1.
3. **CAN bus.**
   - Velocidad de transmisión: 1 Mbps.
4. **Protecciones.**
   - Sobre corriente de 10 A durante 15 s (si una corriente de 10 A o más se prolonga durante 15 s, la protección interviene cortando el suministro de corriente. Aunque al arranque la intensidad de pico es más elevada que 10 A, la protección no interviene debido a que no excede el tiempo máximo de 15 s.

- I2t sobre corriente: 10 A para 30 s (este valor indica que el máximo tiempo que puede operar el driver ISCM8005 con la corriente de 10 A. Este valor es proporcionado por el fabricante del driver).

5. **Parámetros de operación del driver**. (el propio driver lo auto detecta, pulsando en el botón "Detect", marcado en verde en la Figura 62)
   - Tensión de alimentación: 25 V.
   - Limite de corriente: 2 A.
6. **Reguladores.**
   - Control de corriente: (se ajusta de forma automática, pulsando en el botón "Tune & Test", marcado en verde en la Figura 62).
     - Kp: 0.
     - Ki: 0,3916.
   - Control de velocidad: (se ajusta de forma automática, pulsando en el botón "Tune & Test", marcado en verde en la Figura 62).
     - Kp: 185,8.
     - Ki: 16,5.
     - *Integral limit*: 50 %.
     - *Feedforward* (aceleración): 0.
   - Control de posición: (se ajusta de forma automática, pulsando en el botón "Tune & Test", marcado en verde en la Figura 62).
     - Kp: 0,10001.
     - Ki: 0,002502.
     - *Integral limit*: 10 %.
     - *Feedforward* (velocidad): 0.
     - *Feedforward* (aceleración): 0.
     - Límite de velocidad: 468,1 rad/s.
7. **Polaridad de las entradas.**
   - *Enable* (habilitar): Activa a nivel bajo (la tarjeta se habilita después de encenderse).
   - *Limit Switch* + (limite positivo): Activa a nivel bajo.
   - *Limit Switch* – (limite negativo): Activa a nivel bajo.

4.3.3 Programación en TML: Motion

Cuando ya tenemos totalmente configurado el driver, el siguiente paso será el de crear el programa en el leguaje TML. Dicho programa se emplea para establecer un punto de referencia una vez iniciada la aplicación.

Para ello debemos pulsar sobre la opción **Motion,** que se muestra en la siguiente figura:

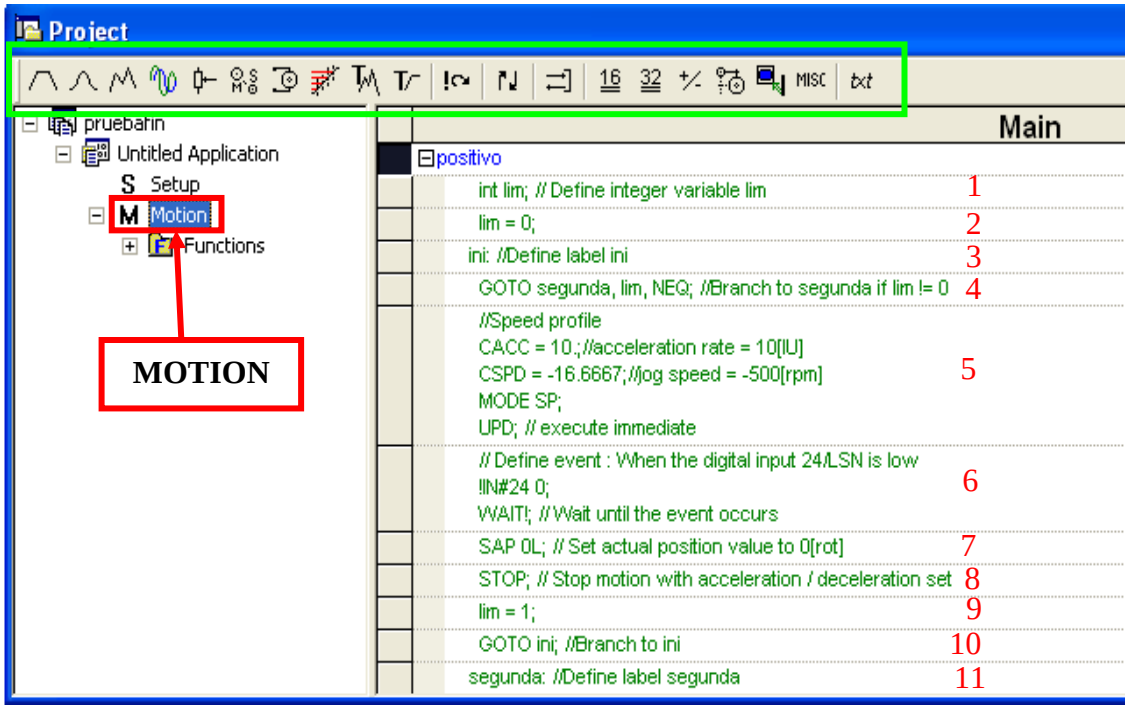


Figura 65: Motion.

A continuación, se expondrá la secuencia creada a partir de funciones propias del lenguaje de movimientos TML. Las opciones disponibles para la creación del programa de movimiento TML, se encuentran en la barra de herramientas (marcada de verde) de la figura anterior.

Se va a proceder a explicar cada una de las funciones utilizadas, que se encuentran numeradas en la figura 65. Para el desarrollo de las mismas se ha empleado la barra de herramienta, que se menciono previamente.

.

1. **Int lim;** // Define la variable "lim" como un entero.

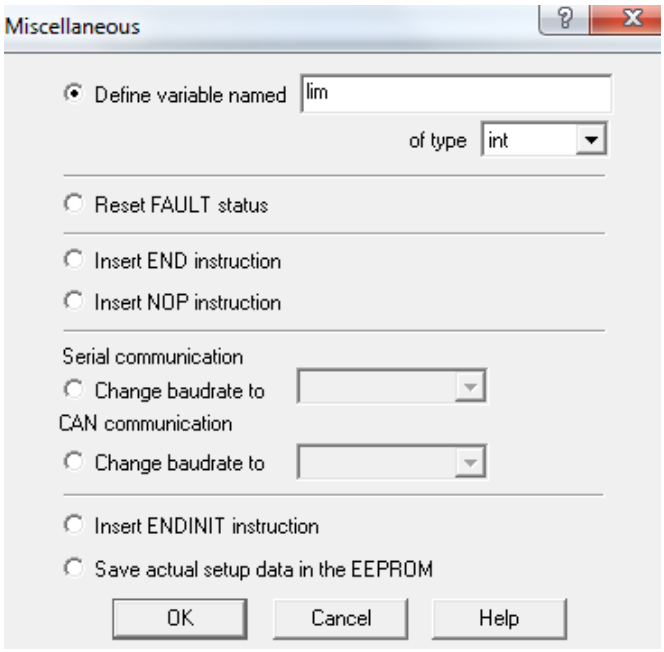


Figura 66: Definición de la variable lim como un entero.

2. **Lim = 0;** // Inicializa la variable "lim" con valor '0'.

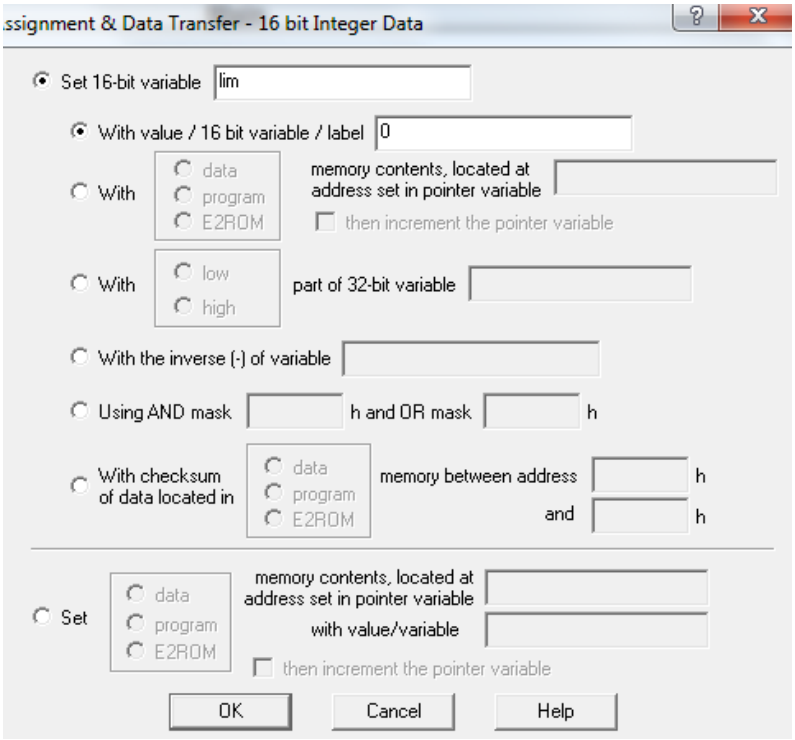


Figura 67: Inicializa la variable lim = 0.

3. **Ini;** // Definir la etiqueta “ini”. Esta etiqueta sirve para indicar el comienzo del bucle.

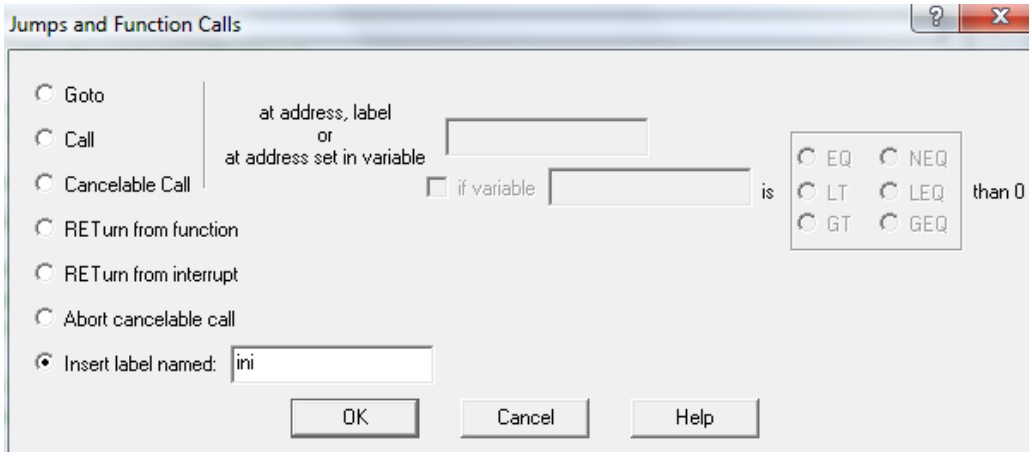


Figura 68: Definir la etiqueta “ini”.

4. **GOTO segunda, lim, NEQ;** // Ir a la etiqueta “segunda” (salta al paso 11) si la variable “lim” es distinta de ‘0’ (lim!=0). Si la variable “lim” es igual a ‘0’ se continúa ejecutando la siguiente acción (continúa con el paso 5).

Figura 69: Ir a la etiqueta “segunda” si la variable “lim” es distinta de ‘0’.

5. // Perfil de velocidad.

   CACC = 10; // Aceleración = 10 [IU].

   CSPD = -16,6667; // Velocidad = -500 [rpm].

   MODE SP; // Modo velocidad.

   UPD; // Ejecución inmediata.

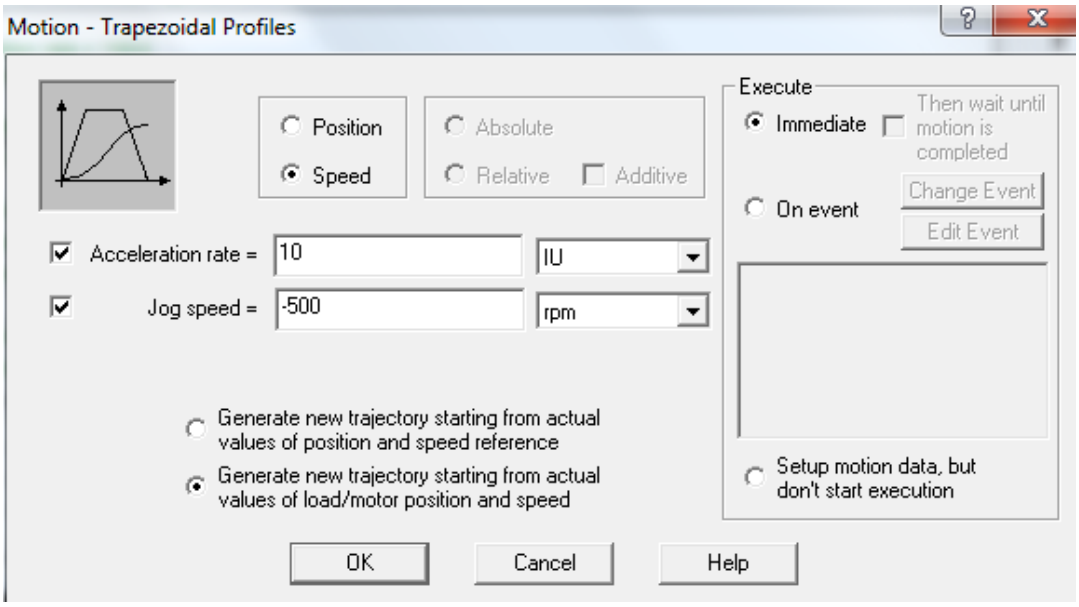


Figura 70: Perfil de velocidad.

6. **Define event;** // Definir evento: cuando la entrada digital LSN (*limit switch negative*) está activa a nivel bajo, es decir, cuando se produzca la activación de la entrada LSN (activa a nivel bajo) se producirá un evento.

   !IN#24 0;

   WAIT; // Esperar hasta que el evento ocurra.

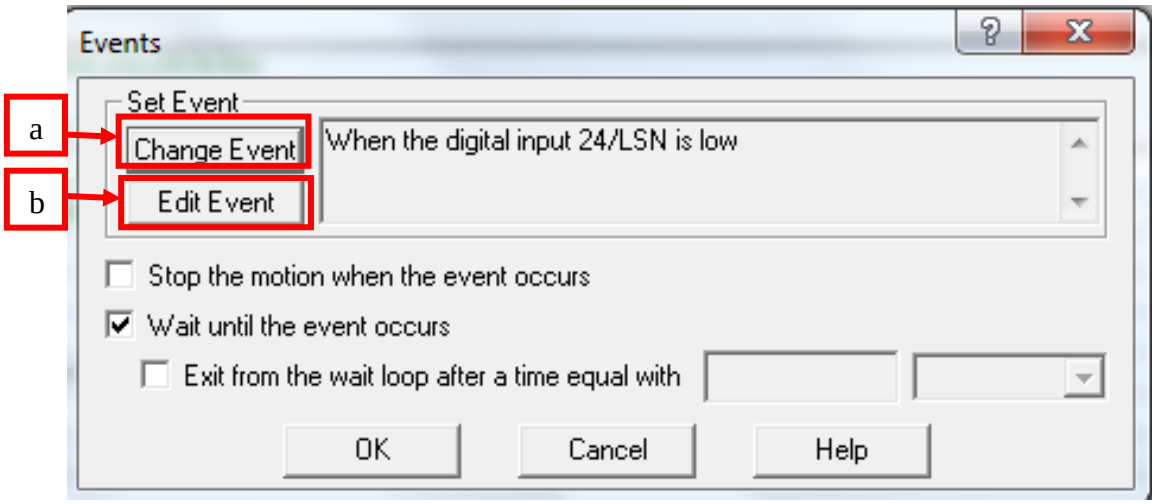


Figura 71: Definir evento.

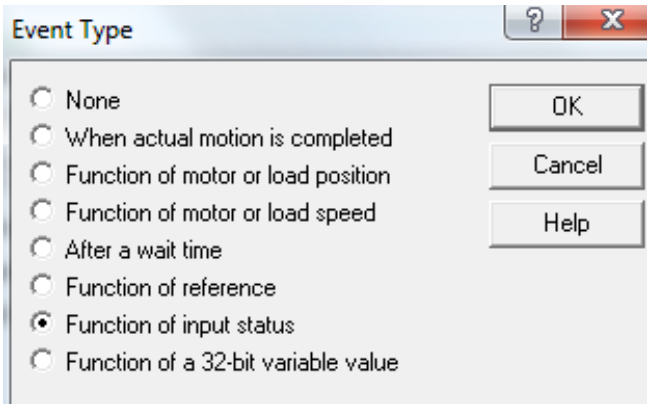


Figura 72: a) Change Event.

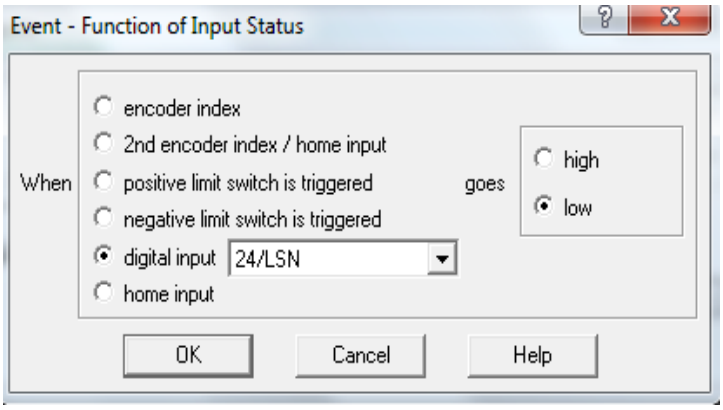


Figura 73: b) Edit Event.

7. **SAP 0L;** // Ajustar la posición actual a '0' [rotaciones].

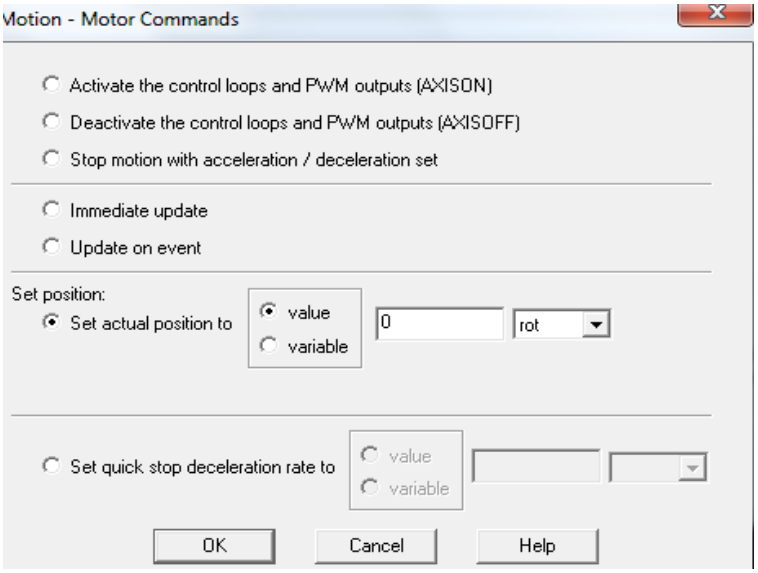


Figura 74: Ajustar la posición actual a '0' rotaciones.

8. **STOP;** // Parar la ejecución del movimiento con aceleración / deceleración.

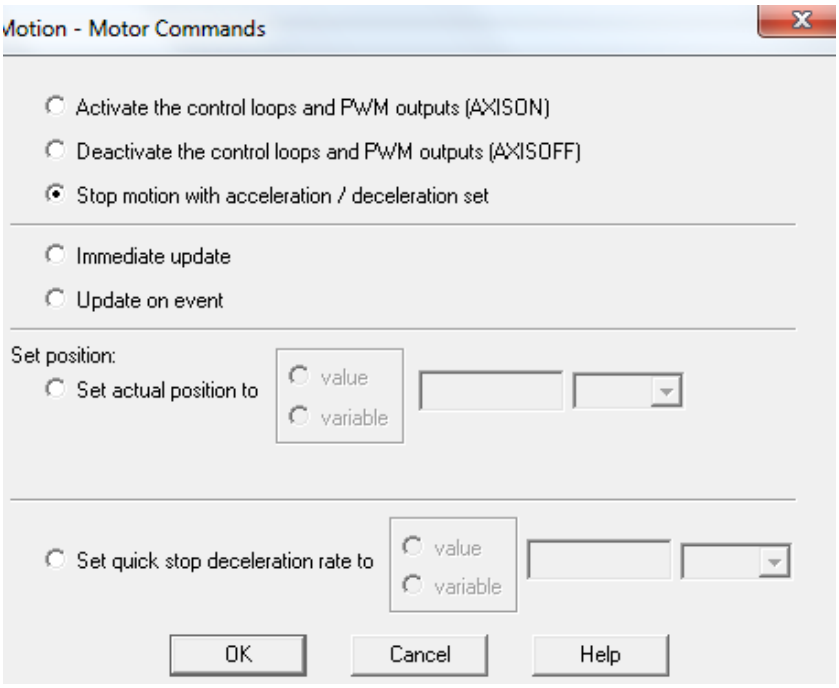


Figura 75: Parar la ejecución del movimiento.

9. **Lim = 1;** // Asignar un '1' en la variable "lim".

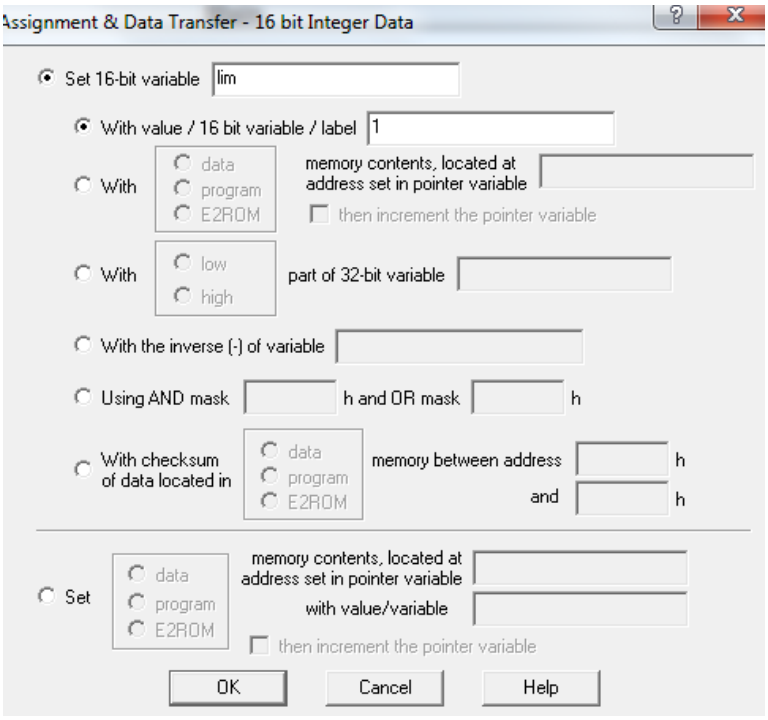


Figura 76: Asignar un '1' en la variable "lim".

10. **GOTO ini;** // Ir a la etiqueta "ini" (volver al paso 3). Esta función crea el bucle entre el paso 10 y el paso 3. Y como se vio antes, es en el paso 4 donde se regresa al final del programa o bien continúa la ejecución de proceso.

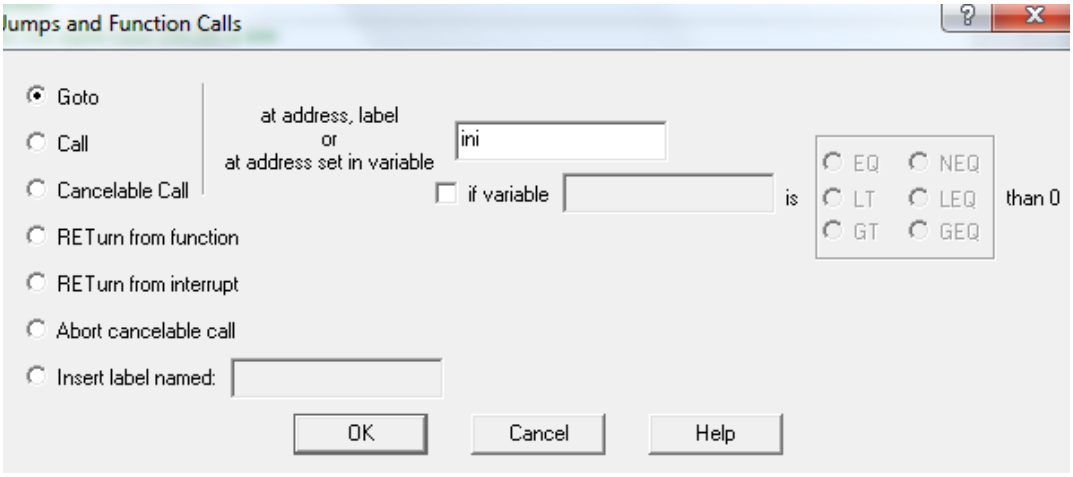


Figura 77: Volver a la etiqueta "ini".

11. **Segunda;** // Definir la etiqueta segunda.

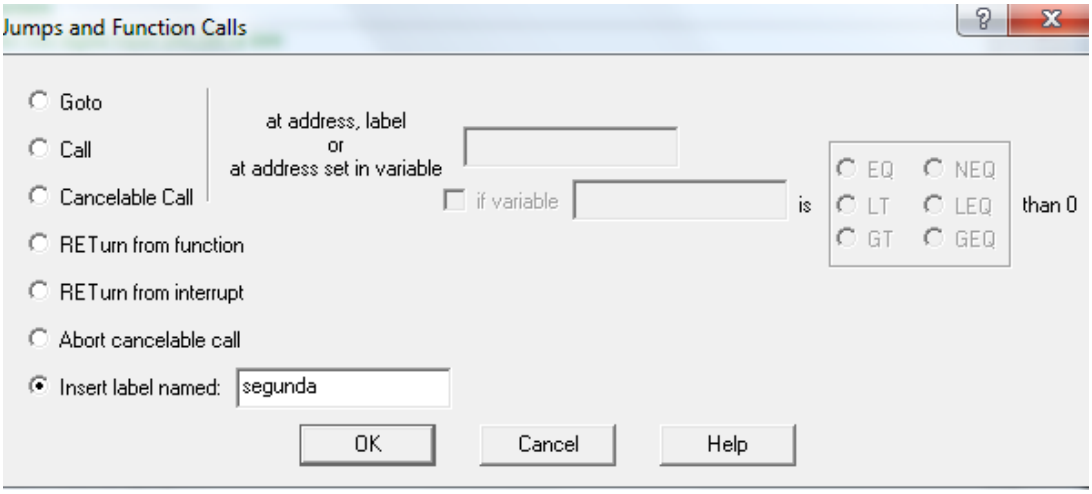


Figura 78: Definir la etiqueta "segunda".

Una vez implementado el programa, se compila la aplicación pulsando "**Aplication/Motion/Build**". Después, se descargará el programa en la memoria interna EEPROM del driver ISCM8005 pulsando "**Aplication/Motion/Download Program"** (se descarga el programa en la memoria temporal)

4.3.4 Grabar en la memoria del driver el programa de TML (EEPROM Programmer)

Para grabar el programa en la memoria permanente del driver, es necesario crear una imagen del programa mediante el uso de EasyMotion Studio. Para crear el programa hay que pulsar en "**Aplication/Create EEPROM Program File/Motion and Setup…".** Es necesario elegir el destino donde se guardará y el nombre del programa con extensión "**.sw**".

Cuando se ha creado el archivo con extensión "**.sw**", se ejecuta el programa EEPROM Programmer de la marca Technosoft.

Lo primero es establecer los parámetros de comunicación necesarios para realizar la programación de la memoria.

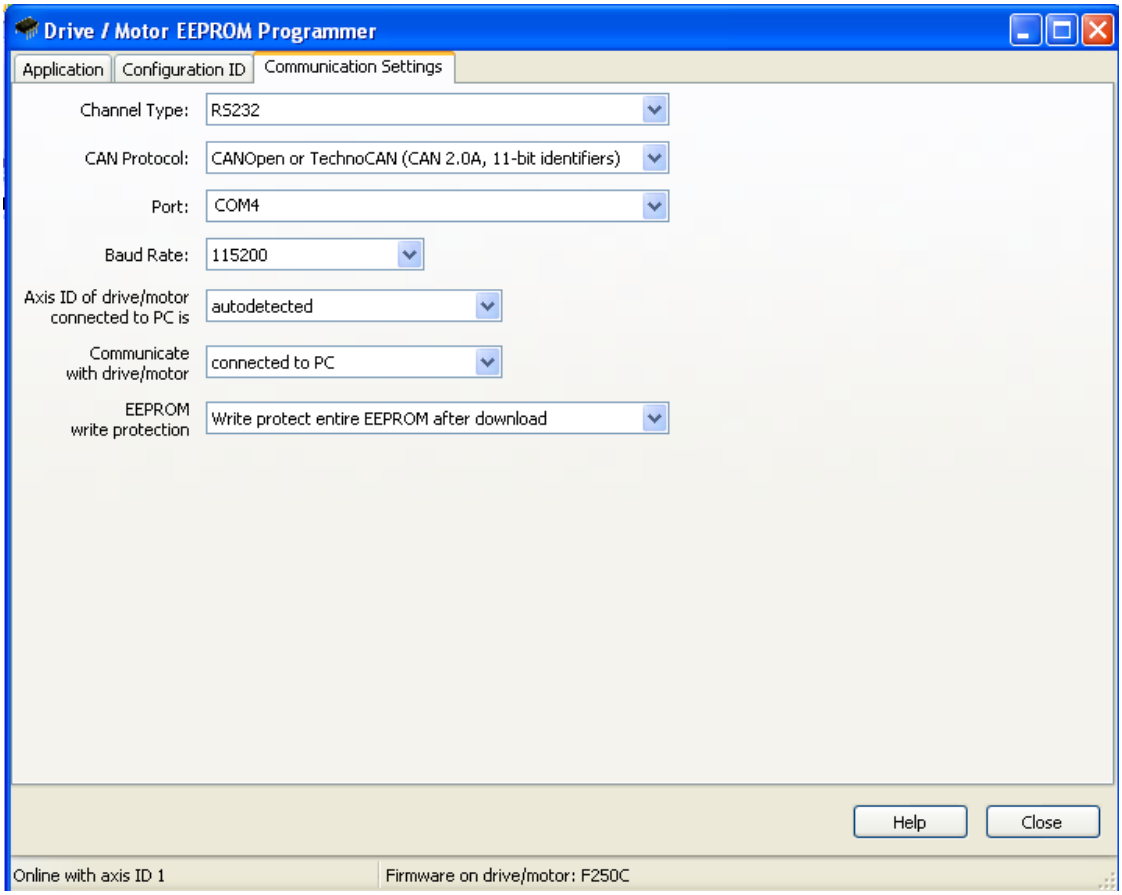


Figura 79: Configuración de la comunicación.

A continuación se detallan los parámetros escogidos:

- Tipo de canal: RS-232.
- Protocolo CAN: CANopen o TechnocCAN.
- Puerto: COM 4.
- Velocidad de transmisión: 115200.
- ID (identificador) de eje: Auto detectar.
- Comunicación con el driver: Conectado a PC.
- Protección de la memoria EEPROM: proteger después de descargar el programa.

El siguiente paso sería realizar la configuración del ID. Esto se detalla en la figura que se muestra a continuación:

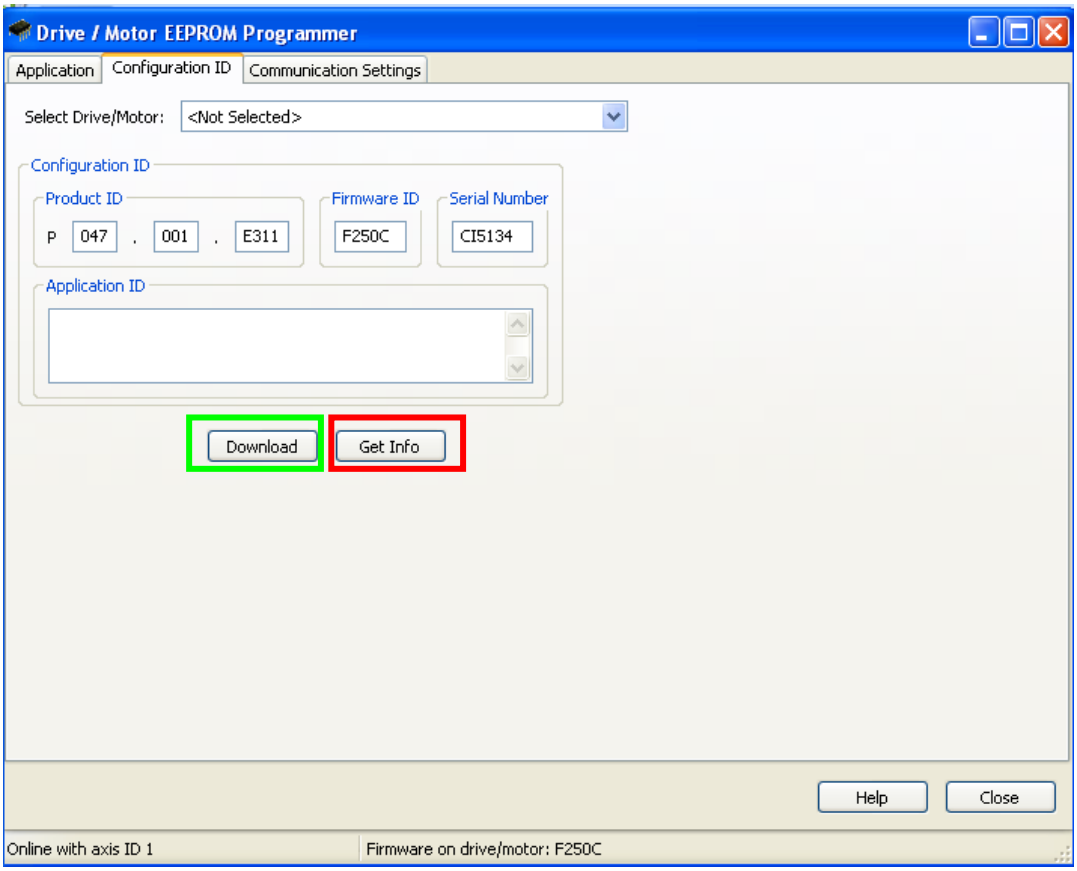


Figura 80: Configuración del ID del driver.

Cuando se selecciona la pestaña de configuración de ID, hay que realizar dos acciones. La primera es pulsar el botón marcado con rojo que aparece en la imagen anterior, una vez hecho esto se toma automáticamente la configuración interna del driver activo. La segunda acción se realiza pulsando el botón marcado con verde. Esto sirve para descargar la configuración seleccionada en el driver.

Por último solo quedaría seleccionar la pestaña "aplicación". Hay que buscar el archivo generado con EasyMotion Studio y ponerlo en el campo "SW File", para ello hay que pulsar el botón marcado en verde. Después pulsar el botón "Download", marcado en rojo, para descargar el fichero en la memoria interna, aparecerá la imagen que se muestra a continuación:

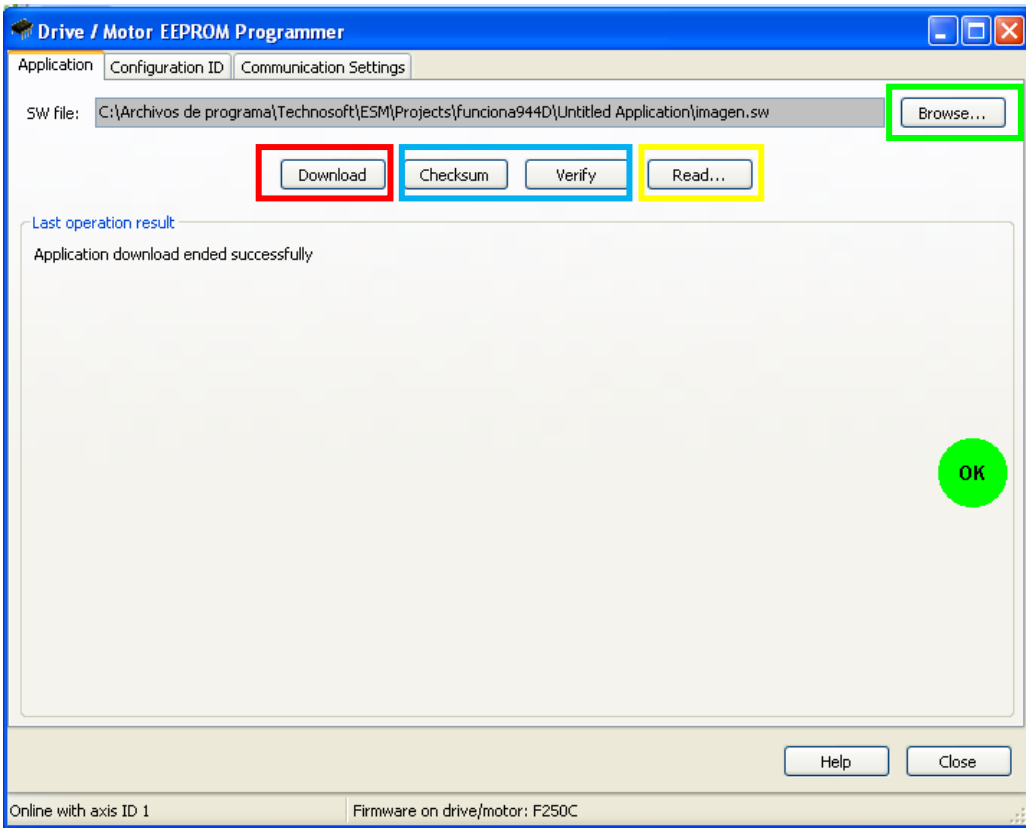


Figura 81: Pestaña aplicación.

Los botones marcados en azul, sirven para verificar que el programa se encuentra en la memoria. El último botón marcado en amarillo, sirve para recopilar el archivo con extensión ".sw" y guardarlo dentro del PC.

4.4 Fase de implementación de las funciones C/C++

Como ya se explico en el primer capítulo, Roboardbot.h es un fichero de tipo cabecera, donde se encuentra declarada la "clase" de C++ que fue el desarrollo principal de este Trabajo de Fin de Grado: RoboardBot. Esta clase hereda de 4 clases de funciones puras virtuales de YARP_dev que agrupan funcionalidades propias del driver (gestión, control en posición, en velocidad, encoders). Las implementaciones se encuentran en ficheros que se llaman igual que las clases base con el sufijo Impl.cpp, que se explicarán a continuación.

4.4.1 IDeviceImpl.cpp

Se encarga de abrir todos los nodos del puerto CAN e inicializar el driver para que se prepare para el proceso [14]. El cuadro que aparece a continuación, muestra el código utilizado para la inicialización del driver mediante comandos "ioctl".

```
node = ::open("/dev/can0",O_RDWR);    // Abrir nodo.
if(node<0)  {
                perror("Couldn't open node.");
                return 1;
            }
cout << "device open (node=" << node << ")" << endl;

if(ioctl(node, IOC_RESET_BOARD)!=0)       // Resetear nodo
            {
                perror("Couldn't reset board.");
                return 1;
            }
cout << "board reset" << endl;

if(ioctl(node, IOC_SET_BITRATE,&bitrate)!=0) // Ajustar velocidad
            {
                perror("Error while setting bitrate.");
                return 1;
            }
cout << "bitrate set" << endl;


if(ioctl(node,IOC_START)!=0)  // Iniciar nodo.
            {
                perror("Couldn't start node");
                return 1;
            }
cout << "node started" << endl;

memset(&message,0,sizeof(struct can_msg));
cout << "startup complete" << endl;
```

A continuación, se muestran los siguientes fragmentos de código donde se emplean mensajes de protocolo CANopen, para llevar a cabo la siguiente parte de la inicialización del driver. Esta inicialización corresponde a la apertura del nodo.

```
bool RoboardBot::open(Searchable& config)
```

Los mensajes que se envían al nodo para iniciarlo, son los siguientes:

```
1       uint8_t msg1[] = {0x01,0x01}; //Start node 1
2       uint8_t msg2[] = {0x06,0x00}; //Ready
3       uint8_t msg3[] = {0x07,0x00}; //Switch on
4       uint8_t msg4[] = {0x0F,0x00}; //Mode: Operation Enable
```

1. Mensaje NMT, dirigido al nodo '1' y con la función de "iniciar el nodo".

```
// 1/4

message.ff=FF_NORMAL;
message.id=0x00;
message.dlc=2;

memcpy(message.data,msg1,2*sizeof(uint8_t));

if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg)){
            perror("Error while sending initialization message (1)");
            return 1;
         }
cout << "msg 1 (Start node) sent" << endl;
```

2. Este mensaje se encarga de dejar el driver en el estado "preparado para encenderse".

```
// 2/4

message.id=0x201+m2; // La variable m2 corresponde a un bucle
memcpy(message.data,msg2,2*sizeof(uint8_t));

if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg)){
            perror("Error while sending initialization message (2)");
            return 1;
         }
cout << "msg 2 (Node ready) sent" << endl;
```

3. Este mensaje se encarga de dejar el driver en el estado “encendido”.

```
// 3/4

memcpy(message.data,msg3,2*sizeof(uint8_t));

if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg)){
            perror("Error while sending initialization message (3)");
            return 1;
          }
cout << "msg 3 (Switch on) sent" << endl;
```

4. Este mensaje se encarga de dejar el driver en el estado “listo para funcionar” (“operación habilitada”).

```
// 4/4

memcpy(message.data,msg4,2*sizeof(uint8_t));

if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg)){
               perror("Error while sending message (4)");
               return 1;
          }
cout << "msg 4 (Operation Enable) sent" << endl;
```

Ahora, se continuara explicando la segunda parte del archivo fuente IDeviceImpl.cpp, que corresponde con el cierre del nodo.

```
bool RoboardBot::close()
```

Los mensajes que se envían al nodo para cerrarlo son los siguientes:

```
1       uint8_t msg3[] = {0x07,0x00}; //Switch on
2       uint8_t msg2[] = {0x06,0x00}; //Ready
```

1. Este mensaje se encarga de cambiar el estado del driver de “operación habilitada” al estado “encendido”.

```
message.id=0x201+m3;
message.dlc=2;

memcpy(message.data,msg3,2*sizeof(uint8_t));

if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg)){
              perror("Error while sending message (8)");
              return 1;
         }

cout << "msg  off 1/2 - sent" << endl;
```

2. Este mensaje se encarga de cambiar el estado del driver de “encendido” al estado “preparado para encenderse”.

```
memcpy(message.data,msg2,2*sizeof(uint8_t));

if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg)){
                 perror("Error while sending message (8)");
                 return 1;
          }
cout << "msg  off 2/2 - sent" << endl;
```

4.4.2 IPositionImpl.cpp

Este archivo fuente contiene funciones relacionadas con el modo de control en posición. Las funciones que se han desarrollado en este archivo fuente, realizan labores tales como establecer el control del driver en modo posición, mover el motor en coordenadas absolutas o relativas, ajustar una velocidad y aceleración de referencia para el movimiento del motor.

A continuación se van comentar las principales funciones implementadas en este archivo fuente.

GetAxes: esta función devuelve el número de ejes a controlar por el sistema.

```
bool RoboardBot::getAxes(int *ax)
 {
  *ax = NUM_MOTORS;
  printf("RoboardBot reporting %d axes are present\n", *ax);
  return true;
 }
```

SetPositionMode: esta función se emplea para establecer en el driver el control, en modo posición.

```
bool RoboardBot::setPositionMode()
{
 printf("RoboardBot: setPositionMode().\n");

 uint8_t msg5[]={0x2F,0x60,0x60,0x00,0x01,0x00,0x00,0x00};

 for(int m4=0;m4<NUM_MOTORS;m4++)
    {
     message.dlc=8;
     message.id=0x601+m4;

     memcpy(message.data,msg5,8*sizeof(uint8_t));

     if( write(node, &message, sizeof(struct can_msg)) <
     sizeof(struct can_msg))
            {
               perror("Error while sending message (5)");
               return 1;
            }
     cout << "msg 5 (Motion type) sent (motor = "<< m4 << ")"<< endl;
    }

 modePosVel = 0;

 return true;
}
```

PositionMove: esta función realiza el movimiento de los motores en coordenadas absolutas.

```
bool RoboardBot::positionMove(int j, double ref)
{
 if(modePosVel!=0)
       {
        printf("RoboardBot: Not in position mode.\n");
        setPositionMode();
        return false;
       }
 else printf("RoboardBot::positionMove(%d,%f) \n",j,ref);

 if( (j<0) || (j>10) )
       {
        printf("RoboardBot: Introduce un valor entre [0-10].\n");
        return false;
       }
 else printf("RoboardBot::positionMove(%d,%f) [begin]\n",j,ref);

 uint8_t msgPosition[]={0x23,0x7A,0x60,0x00,0x00,0x00,0x00,0x00};
 uint8_t msgRun[]={0x1F,0x00};

 int position = (int)((4*ref*500*250)/360); //De [SI] a [IU].

 memcpy(msgPosition+4,&position,4);

 message.ff=FF_NORMAL;
 message.id=0x601+j;
 message.dlc=8;

 memcpy(message.data,msgPosition,8*sizeof(uint8_t));

 if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg))
       {
        perror("Error while sending msg_position (6)");
        return 1;
       }

 cout << "Target position msg_position sent";
 CAN::display(message);

 message.id=0x201+j;
 message.dlc=2;

 memcpy(message.data,msgRun,2*sizeof(uint8_t));

 if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg))
       {
        perror("Error while sending msg_run (8)");
        return 1;
       }

 cout << "Start movement msg_run sent";
 CAN::display(message);

 stop(j);
 joint_status[j]=1;
 printf("RoboardBot::positionMove(%d,%f) [end]\n",j,ref);
 return true;
}
```

RelativeMove: esta función realiza el movimiento de los motores en coordenadas relativas.

```
bool RoboardBot::relativeMove(int j, double delta)
{
 if(modePosVel!=0)
      {
       printf("RoboardBot: Not in position mode.\n");
       setPositionMode();
       return false;
      }
 else printf("RoboardBot::relativeMove(%d,%f) \n",j,delta);

if( (j<0) || (j>10) )
      {
       printf("RoboardBot: Introduce un valor entre [0-10].\n");
       return false;
      }
 else printf("RoboardBot::relativeMove(%d,%f) \n",j,delta);

 uint8_t msgPosition2[]={0x23,0x7A,0x60,0x00,0x00,0x00,0x00,0x00};
 uint8_t msgRun2[]={0x5F,0x00};


 target_degrees[j]=(updated_degrees[j]+delta);
 int posrel =(int)((target_degrees[j]*250*4*500)/360); //De [SI]a[IU]
 sendVel[j] = SPEED_ADJ*refVel[j];

 memcpy (msgPosition2+4,&posrel,4);

 message.ff=FF_NORMAL;
 message.id=0x601+j;
 message.dlc=8;

 memcpy(message.data,msgPosition2,8*sizeof(uint8_t));

 if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg))
      {
       perror("Error while sending msg_position2 (9)");
       return 1;
      }
 cout << "Target position msg_position2 sent";
 CAN::display(message);

 message.id=0x201+j;
 message.dlc=2;

 memcpy(message.data,msgRun2,2*sizeof(uint8_t));

 if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg))
      {
       perror("Error while sending msg_run (11)");
       return 1;
      }
 cout << "Start movement msg_run2 sent";
 CAN::display(message);

 stop(j);

 joint_status[j]=2;
 printf("RoboardBot::relativeMove(%d,%f) [end]\n",j,delta);
 return true;
}
```

PositionMove: es la misma función pero recibe parámetros diferentes a la función anterior de mismo nombre. Esta función llama tantas veces a la función "positionMove (int j, doublé ref)" como motores se estén controlando.

```
bool RoboardBot::positionMove(const double *refs)
{
 for(int motor=0;motor<NUM_MOTORS;motor++)
        positionMove(motor, refs[motor]);
 return true;
}
```

RelativeMove: es la misma función pero recibe parámetros diferentes a la función anterior de mismo nombre. Esta función llama tantas veces a la función "relativeMove (int j, doublé delta)" como motores se estén controlando.

```
bool RoboardBot::relativeMove(const double *deltas)
{
 for(int motor=0;motor<NUM_MOTORS;motor++)
      relativeMove(motor, deltas[motor]);
 return true;
}
```

SetRefSpeed: esta función establece una velocidad ("sp") de referencia en el nodo "j". Esta velocidad es necesaria para poder realizar el movimiento requerido.

```
bool RoboardBot::setRefSpeed(int j, double sp)
{
 if( (j<0) || (j>10) )
       {
        printf("RoboardBot: Introduce un valor entre [0-10].\n");
        return false;
       }
 else printf("RoboardBot::SetRefSpeed(%d,%f) [begin]\n",j,sp);

 uint8_t msg7[]={0x23,0x81,0x60,0x00,0x00,0x00,0x05,0x00};

 double velocity3 = ((4*500*250*0.001*sp)/60);      //De [SI] a [IU]
 int speed3 = (int) (velocity3*65536);

 memcpy(msg7+4,&speed3,4);       //Comentado por Roberto 16/7/2012

 message.id=0x601+j;
 message.dlc=8;

 memcpy(message.data,msg7,8*sizeof(uint8_t));

 if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg))
       {
        perror("Error while sending message (7)");
        return 1;
       }
 cout << "msg 7 (Target speed) sent" << endl;
 return true;
}
```

SetRefSpeeds: es la misma función pero recibe parámetros diferentes a la función anterior de mismo nombre. Esta función llama tantas veces a la función "setRefSpeed (int j, doublé sp)" como motores se estén controlando.

```
bool RoboardBot::setRefSpeeds(const double *spds)
{
 for (unsigned int i=0; i<NUM_MOTORS; i++)
      {
       setRefSpeed(i,spds[i]);
      }
 return true;
}
```

SetRefAcceleration: esta función establece una aceleración ("acc") de referencia en el nodo "j". Esta aceleración es necesaria para poder realizar el movimiento requerido.

```
bool RoboardBot::setRefAcceleration(int j, double acc)
{
 if( (j<0) || (j>10) )
      {
       printf("RoboardBot: Introduce un valor entre [0-10].\n");
       return false;
      }
 else printf("RoboardBot::SetRefAcceleration(%d,%f) \n",j,acc);

 uint8_t msg_acc[]={0x23,0x83,0x60,0x00,0xFF,0x03,0x00,0x00};

 message.id=0x601+j;
 message.dlc=8;

 memcpy(message.data,msg_acc,8*sizeof(uint8_t));

 if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg))
      {
       perror("Error while sending message acc");
       return 1;
      }
 cout << "msg_acc (Target acc) sent" << endl;

 refAcc[j]=acc;
 return true;
}
```

SetRefAccelerations: esta función recibe parámetros diferentes a la función anterior de mismo nombre. Establece un nuevo parámetro de aceleración para la referencia de aceleración. Se establecen tantas referencias, como motores se estén controlando.

```
bool RoboardBot::setRefAccelerations(const double *accs)
{
 for (unsigned int i=0; i<NUM_MOTORS; i++)
      refAcc[i]=accs[i];
 return true;
}
```

Stop: esta función se encarga de parar el nodo. Siendo necesaria su implementación, ya que después de la ejecución de ciertas funciones como: "positionMove" o "relativeMove", se tiene que dar la orden de parar la ejecución de la posición demandada, para que se pueda ejecutar una nueva posición.

```
bool RoboardBot::stop(int j)
{
 if( (j<0) || (j>10) )
      {
        printf("RoboardBot: Introduce un valor entre [0-10].\n");
        return false;
      }
 else printf("RoboardBot::Stop(%d) [begin]\n",j);


 uint8_t msg_stop[]={0x0F,0x00};

 message.id=0x201+j;
 message.dlc=2;

   memcpy(message.data,msg_stop,2*sizeof(uint8_t));

 if( write(node, &message, sizeof(struct can_msg)) < sizeof(struct
can_msg))
      {
       perror("Error while sending message stop");
       return 1;
      }
 cout << "msg_stop (stop) sent" << endl;

 printf("RoboardBot::stop(%d) [end]\n",j);

 return true;
}
```

4.4.3 IVelocityImpl.cpp

Este archivo fuente contiene funciones relacionadas con el modo de control en velocidad. Las funciones que se han desarrollado en este archivo fuente, realizan labores tales como establecer el control del driver en modo velocidad, mover el motor en modo velocidad, ajustar una velocidad y aceleración para llevar a cabo el movimiento del motor.

A continuación se van comentar las principales funciones implementadas en este archivo fuente.

SetVelocityMode: esta función se emplea para establecer en el driver el control, en modo velocidad.

```
bool RoboardBot::setVelocityMode()
 {
  for (int m=0;m<NUM_MOTORS;m++)
      {
       printf("RoboardBot::setVelocityMode() [begin]\n");

       uint8_t msg6[]={0x2F,0x60,0x60,0x00,0x03,0x00,0x00,0x00};

       message.dlc=8;
       message.id=0x601+m;

       memcpy(message.data,msg6,8*sizeof(uint8_t));

       if(write(node,&message,sizeof(struct can_msg)) < sizeof(struct
       can_msg))
            {
             perror("Eror while sending message (15)");
             return 1;
            }
      cout<<"msg6 (Motion type) sent (motor = "<< m <<" ) " << endl;
      CAN::display(message);
     }
  modePosVel=1;
  printf("RoboardBot::setVelocityMode() [end]\n");
  return true;
 }
```

VelocityMove: esta función realiza el movimiento de los motores en modo velocidad. Los parámetros que recibe esta función son: el número de nodo "j" y la velocidad a la que se debe mover el motor.

```
bool RoboardBot::velocityMove(int j, double sp)
 {
  if (modePosVel!=1)
      {
       printf("RoboardBot: Not in velocity mode.\n");
       setVelocityMode();
       return false;
      }
  else printf("RoboardBot:: velocityMove(%d,%f) \n",j,sp);

  if ( (j<0) || (j>10) )
      {
       printf("RoboardBot:Introduce un valor entre [0-10].\n");
       return false;
      }
  else printf("RoboardBot::VELOCITYMOVE(%d,%f) \n",j,sp);


  uint8_t msg7[]={0x23,0xFF,0x60,0x00,0x00,0x00,0x00,0x00};

  double velocity2 = ((4*500*250*0.001*sp)/60);  //De [SI] a [IU].
  int speed2 = (int) (velocity2*65536);

  memcpy(msg7+4,&speed2,4);

  message.dlc=8;
  message.id=0x601+j;

  memcpy(message.data,msg7,8*sizeof(uint8_t));

  if (write(node,&message,sizeof(struct can_msg))< sizeof(struct
can_msg))
      {
       perror ("Error while sending message 7 (16)");
       return 1;
      }
  cout << "msg7 (velocityMove) sent (motor = "<< j <<" ) "<< endl;
  CAN::display(message);

  canBusReady.post();
  joint_status[j]=3;
  return true;
 }
```

VelocityMove: esta función se utiliza para mover varios motores de forma simultánea, ejecuta tantas veces la función "velocityMove (int j, doublé sp)" como motores se estén controlando.

```
bool RoboardBot::velocityMove(const double *sp)
 {
  for (unsigned int i=0; i<NUM_MOTORS; i++)
        velocityMove(i,sp[i]);
  return true;
 }
```

4.4.4 IEncoderImpl.cpp

Este archivo fuente contiene funciones relacionadas con control de los sensores de posición (encoder). Las funciones que se han desarrollado en este archivo fuente, realizan labores tales como resetear el encoder, detectar la velocidad o posición del motor.

A continuación se van comentar las principales funciones implementadas en este archivo fuente.

ResetEncoder: esta función se encarga de ejecutar el programa TML. Cuando se inicia el driver se ejecuta esta función, por lo cual se establece un sistema de coordenadas al comienzo del programa.

```
bool RoboardBot::resetEncoder(int j)
 {
  uint8_t msgEncReset[]={0x2B,0x77,0x20,0x00,0x01,0x00,0x00,0x00};
  message.ff=FF_NORMAL;
  message.id=0x601+j;
  message.dlc=8;

  memcpy(message.data,msgEncReset,8*sizeof(uint8_t));

  if( write(node,&message,sizeof(struct can_msg))< sizeof(struct
can_msg))
    {
     perror(" Error while sending msgEncReset ");
     return 1;
    }
  cout<< "Call fuction TML --- msgEncReset sent"<<endl;
  CAN::display(message);
  return true;
 }
```

ResetEncoders: esta función se utiliza para resetear varios motores de forma simultánea, ejecuta tantas veces la función “resetEncoder (int j)” como motores se estén controlando.

```
bool RoboardBot::resetEncoders()
 {
  for(int l=0;l<NUM_MOTORS;l++)
        resetEncoder(l);

  return true;
 }
```

GetEncoder: esta función se utiliza para detectar la posición actual del motor. Los parámetros que se utilizan son el numero de nodo "j", y el puntero "*v" que es donde se almacena el valor de la posición.

```
bool RoboardBot::getEncoder(int j, double *v)
 {
  int k=1;
  int recive;

  uint8_t msgEncoder[8]= {0x40,0x64,0x60,0x00,0x00,0x00,0x00,0x00};

  message.dlc=8;
  message.id=0x601+j;

  memcpy(message.data,msgEncoder,8*sizeof(uint8_t));

  if( write(node, &message, sizeof(struct can_msg))< sizeof(struct
can_msg))
      {
       perror("Error while sending msg_Encoder (20)");
       return 1;
      }
  cout << " msg_Encoder sent " << endl;
  CAN::display(message);
  while (k==1)
      {
       int res = CAN::read_timeout(node,&message_rec,2000);
       cout << "read_timeout returned " << res << endl;

       if (res <=  0) return false;
       if (message_rec.data[1]==100)
            {
             if(message_rec.data[2]==96)
                 k=0;
            }
      }
  CAN::display(message_rec);
  memcpy(&recive,message_rec.data+4,4);
  cout << "got " << recive << endl;

  *v = ((recive*360)/(4*500*250));    //De [IU] a [SI].
  return true;
 }
```

GetEncoders: esta función se utiliza para detectar la posición de varios motores de forma simultánea, ejecuta tantas veces la función "getEncoder (int j, doublé *v)" como motores se estén controlando.

```
bool RoboardBot::getEncoders(double *encs)
 {
  double degrees;
  for (unsigned int i=0; i<NUM_MOTORS; i++)
            getEncoder(i,&encs[i]);
  return true;
 }
```

GetEncoderSpeed: esta función se utiliza para detectar la velocidad actual del motor. Los parámetros que se utilizan son el numero de nodo "j", y el puntero "*sp" que es donde se almacena el valor de la velocidad.

```
bool RoboardBot::getEncoderSpeed(int j, double *sp)
 {
  int k2=1;
  int speedrecive;

uint8_t msgEncoderspeed[8]={0x40,0x6C,0x60,0x00,0x00,0x00,0x00,0x00};

  message.dlc=8;
  message.id=0x601+j;

  memcpy(message.data,msgEncoderspeed,8*sizeof(uint8_t));

  if( write(node, &message, sizeof(struct can_msg))< sizeof(struct
can_msg))
      {
       perror("Error while sending msg_Encoder (20)");
       return 1;
      }
  cout << " msg_Encoder sent " << endl;
  CAN::display(message);
  while (k2==1)
      {
       int res2 = CAN::read_timeout(node,&message_rec2,2000);
       cout << "read_timeout returned [2] :  " << res2 << endl;
       if (res2 <=  0) return false;
       if (message_rec2.data[1]==108)
            {
             if(message_rec2.data[2]==96)
                        k2=0;
            }
      }
  CAN::display(message_rec2);
  memcpy(&speedrecive,message_rec2.data+4,4);
  cout << "got [2] :  " << speedrecive << endl;

  *sp = ((speedrecive*60)/(4*500*250*0.001*65536)); //De [IU] a [SI].
  return true;
 }
```

GetEncoderSpeeds: esta función se utiliza para detectar la velocidad de varios motores de forma simultánea, ejecuta tantas veces la función "getEncoderSpeed (int j, doublé *sp)" como motores se estén controlando.

```
bool RoboardBot::getEncoderSpeeds(double *spds)
 {
  for (unsigned int i=0; i<NUM_MOTORS; i++)
            getEncoderSpeed(i,&spds[i]);
  return true;
 }
```

4.4.5 Prueba de la aplicación

Un detalle a tener en cuenta, es el sistema operativo utilizado para implementar la aplicación, que es el Ubuntu 11.10. Para poder realizarse la prueba de los programas explicados anteriormente, es necesario llevar a cabo una serie de pasos:

1. Debe realizarse la configuración de la red Ethernet, entre el PC y la CPU RoBoard RB-100. Para ello se debe seleccionar el icono marcado en verde y después hacer clic en "**Editar las conexiones…**" como se muestra en la siguiente figura.

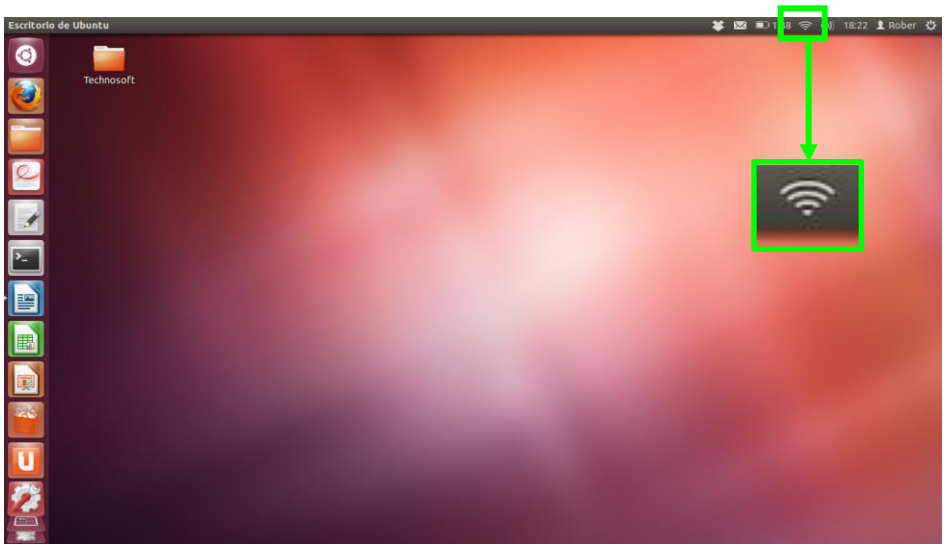


Figura 82: Editar las conexiones.

Una vez hecho lo anterior, aparecerá una ventana como la que se muestra a continuación, en la que hay que elegir el tipo de conexión empleada, que en nuestro caso el tipo es: "**Cableada"** (marcado en verde)**.**

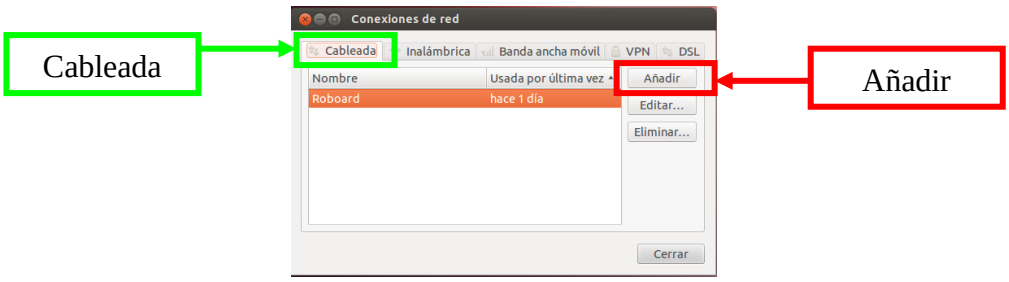


Figura 83: Conexiones de red.

Después de elegir el tipo de conexión, se pulsa sobre el botón "**Añadir**" para crear una nueva configuración, como se muestra en la figura anterior marcado en rojo.

Por último se muestra la siguiente figura donde se establecen el nombre de la red (Roboard) y los parámetros de dirección, más concretamente "**Ajustes de IPv4**" (marcado en rojo, es un tipo de protocolo de seguridad). El método usado es "**manual**", aparece marcado en verde en la siguiente figura. Es necesario establecer tres parámetros básicos (marcados en azul), para configurar la red de forma manual, que son: **Dirección, Mascara de red** y **Puerta de enlace.**

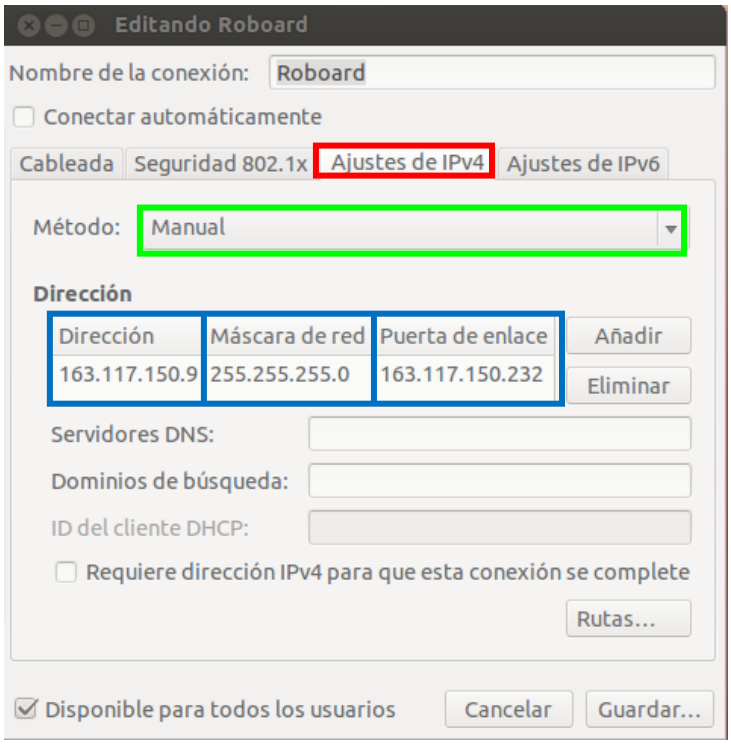


Figura 84: Editando Roboard.

2. Una vez configurada la red, procedemos a comprobar que la conexión funciona correctamente, esto se comprueba desde la terminal. A la terminal se accede mediante los comandos "**Ctrl + Alt + t**". cuando ya está abierta la consola tecleamos el siguiente código: **ping 163.117.150.232**. Aparecerá el siguiente fragmento de código si todo funciona correctamente.

```
robertog@robolb:~$ ping 163.117.150.232
PING 163.117.150.232 (163.117.150.232) 56(84) bytes of data.
64 bytes from 163.117.150.232: icmp_seq=1 ttl=64 time=0.290 ms
64 bytes from 163.117.150.232: icmp_seq=2 ttl=64 time=0.276 ms
64 bytes from 163.117.150.232: icmp_seq=3 ttl=64 time=0.282 ms
^C
--- 163.117.150.232 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.276/0.282/0.290/0.020 ms
```

3. Si la red se encuentra disponible, entonces se puede conectar el PC a la CPU de forma segura, utilizando la conexión cifrada "**SSH**". Para conectarnos a la IP de la puerta de enlace, tecleamos el siguiente código:
   ssh roboard@163.117.150.232. Clave: **roboard.** Aparecerá el siguiente fragmento de código si todo funciona correctamente.

```
robertog@robolb:~$ ssh roboard@163.117.150.232
roboard@163.117.150.232's password:   roboard
Linux rb-100 2.6.28-6-386 #20-Ubuntu Fri Apr 17 08:32:39 UTC 2009
i586

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by
applicable law.

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/
Last login: Wed Apr 11 16:42:56 2012
-bash: [: demasiados argumentos
```

4. Ya se ha establecido la comunicación entre el PC y la CPU RoBoard RB-100. Lo siguiente que hay que realizar sería instalar los controladores software de la tarjeta de comunicación HICO.CAN-miniPCI. Esto se realiza con el siguiente código: **cd hicoCan/driver.** Una vez dentro de la carpeta del controlador software, se realiza la instalación por medio de siguiente código: **sudo make install.** Aparecerá el siguiente fragmento de código si todo funciona correctamente.

```
roboard@rb-100:~/ $ cd hicoCan/driver
roboard@rb-100:~/hicoCan/driver$ sudo make install
rmmod hcanpci 2>/dev/null; \
        insmod hcanpci.ko irqtrace=0 fw_update=0 && \
        ./makenodes.sh
```

5. Para comprobar que los controladores de la tarjeta de comunicación se han instalado correctamente, hay que introducir el siguiente código: **ls /dev/can***. Aparece el siguiente fragmento de código si todo funciona correctamente.

```
roboard@rb-100:~/hicoCan/driver$ ls /dev/can*
/dev/can0  /dev/can1
```

6. Lo siguiente acción a realizar depende de la fase en la que se encuentre la aplicación, si se requiere **editar** la aplicación, o bien, si se requiere **probar** la aplicación. Para **editar** la aplicación se debe seguir la siguiente ruta para acceder a la carpeta "**roboardbot**" donde se editará el programa mediante un editor de texto de Ubuntu 11.10, que se llama **nano.** La ruta de acceso es la que sigue a continuación: **cd puncion/src/libraries/RbPlugins/roboardbot.** Una vez en la carpeta "**roboardbot**", usando **nano + nombre del archivo (.cpp o .h)** se podrán editar los programas. Después de **editar** un archivo es necesario compilarlo. Para realizar la compilación hay que acceder a la siguiente ruta de acceso: **cd puncion/build.** una vez aquí, tecleamos el comando **make,** lo cual compilara todos los archivos que hayan sufrido alguna modificación. Para **probar** la aplicación se tiene que acceder a la siguiente ruta: **cd puncion/build.** Para ejecutar el programa introducimos el siguiente código: **bin/testRoboardBot.** Aparece el siguiente fragmento de código si todo funciona correctamente.

```
roboard@rb-100:~$ cd puncion/build/
roboard@rb-100:~/puncion/build$ bin/testRoboardBot
||| policy set to PUNCION_ROOT
||| environment variable PUNCION_ROOT not set
||| loading policy from /etc/PUNCION_ROOT.ini
yarp: cannot read from /etc/PUNCION_ROOT.ini
||| failed to load policy from /etc/PUNCION_ROOT.ini
Run "testRoboardBot --help" for options.
testRoboardBot checking for yarp network... [ok]
==============================
check our device can be accessed
ControlBoard subdevice is roboardbot
RoboardBot::open [begin]
device open (node=3)
board reset
bitrate set
node started
startup complete
msg 1 (Start node) sent
msg 2 (Node ready) sent
msg 3 (Switch on) sent
msg 4 (Operation Enable) sent
RoboardBot: setPositionMode().
msg 5 (Motion type) sent (motor = 0)
RoboardBot::SetRefSpeed(0,300.000000) [begin]
msg 7 (Target speed) sent
RoboardBot::SetRefAcceleration(0,32767.000000) [begin]
msg_acc (Target acc) sent
Call fuction TML --- msgEncReset sent
        ID: 601 |  Data: 2B - 77 - 20 - 0 - 1 - 0 - 0 - 0 -
RoboardBot::open [end]
yarp: created device <roboardbot>. See C++ class RoboardBot for
documentation.
yarp: Port /roboardbot/rpc:i active at tcp://163.117.150.232:10008
yarp: Port /roboardbot/command:i active at tcp://163.117.150.232:10009
yarp: Port /roboardbot/state:o active at tcp://163.117.150.232:10010
RoboardBot reporting 1 axes are present
RoboardBot reporting 1 axes are present
RoboardBot reporting 1 axes are present
Server control board starting
yarp: created wrapper <controlboard>. See C++ class ServerControlBoard for
documentation.
```

7. El último paso a realizar, se hace desde una nueva ventana del terminal. Cuando disponemos de esta nueva ventana, lanzamos el controlador de los motores con la conexión de tipo **YARP.** Gracias a este tipo de conexión que sirve para comunicación entre programas, se consigue la ejecución de las funciones implementadas anteriormente en este capítulo. Para realizar el control desde con **YARP**, es necesario teclear el siguiente código:
   yarp rpc /roboardbot/rpc:i. Además hay que introducir **help** que nos aporta la interfaz del programa como se muestra a continuación. Aparece el siguiente fragmento de código si todo funciona correctamente.

```
robertog@robolb:~$ yarp rpc /controlboard/rpc:i
help
Responses:
  [help]
  [help] [more]
  [get] [axes]
  [get] [name] $iAxisNumber
  [set] [pos] $iAxisNumber $fPosition
  [set] [rel] $iAxisNumber $fPosition
  [set] [vmo] $iAxisNumber $fVelocity
  [get] [encs]
  [get] [enc] $iAxisNumber
  [set] [poss] ($f0)
  [set] [rels] ($f0)
  [set] [vmos] ($f0)
  [set] [aen] $iAxisNumber
  [set] [adi] $iAxisNumber
  [get] [acu] $iAxisNumber
  [get] [acus]
  ok
```

Un ejemplo de cómo ejecutar las funciones es el siguiente:

```
set pos 0 360
Response: [ok]
set rel 0 520
Response: [ok]
get enc 0
Response: [is] enc 879.0 [ok]
set vmo 0 500
Response: [ok]
```

4.4.6 Interfaz de la aplicación

Para facilitar el manejo de la aplicación por parte del usuario, se ha desarrollado una interfaz via web, con la que se podrá controlar y monitorizar el soporte del robot asistencial ASIBOT. A esta interfaz se puede acceder desde cualquier PC o PDA con conexión a internet.

A continuación se muestra un pantallazo de la interfaz desarrollada:

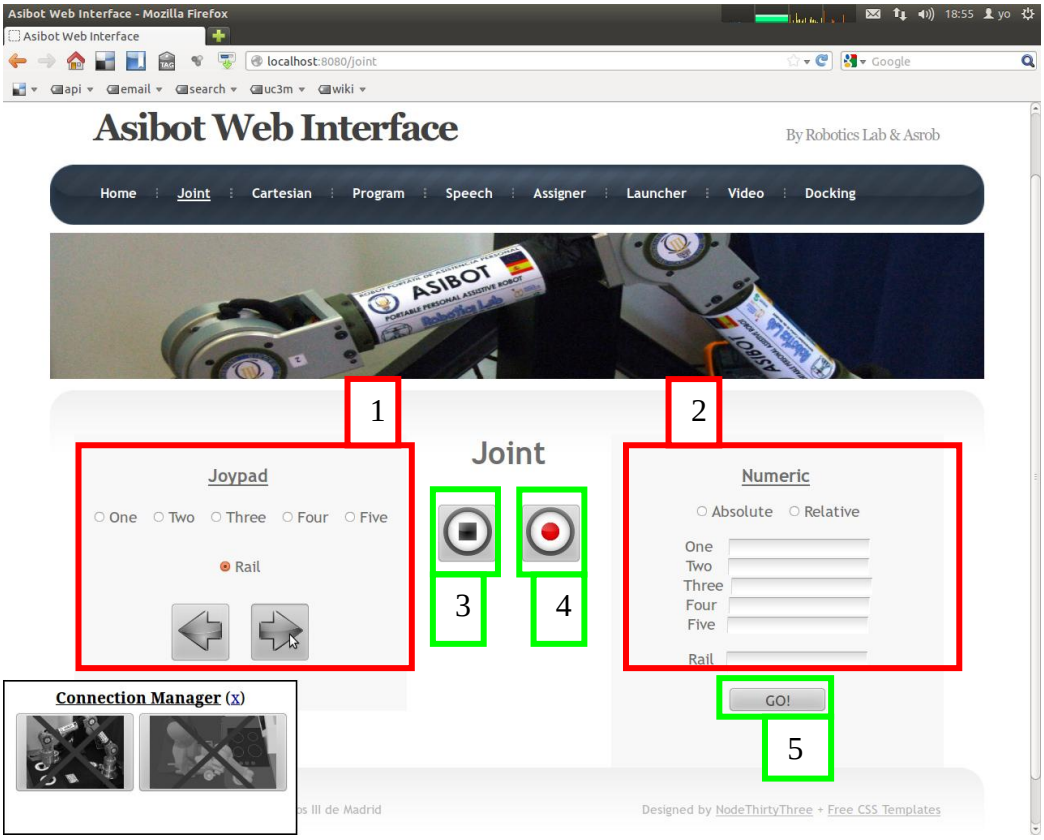


Figura 85: Interfaz de la aplicación.

1. **Movimiento incremental**: se mueve el motor del rail o bien uno de los cinco motores del robot asistencial ASIBOT, de forma incremental (con pequeños incrementos de posición)
2. **Movimiento numérico**: se mueve el motor del rail o bien uno de los cinco motores del robot asistencial ASIBOT, de forma numérica (por medio comandos) pudiéndose ajustar a valores mas precisos que en el modo anterior.

3. **Parar la ejecución**.
4. **Grabar la posición actual**: una vez pulsado dicho botón, aparece ventana donde se introduce el nombre del punto que se introduce.
5. **Ejecutar movimiento**: pulsando dicho botón se ejecutan los comandos especificados en los apartados anteriores.

Capitulo 5: Conclusiones y Trabajos futuros

5.1 Conclusiones

Se han cumplido los objetivos que fueron planteados para este proyecto:

- Se ha implementado un control directo que facilitará el desarrollo de futuros algoritmos de control para el soporte del robot asistencial ASIBOT.
- Se ha desarrollado un software que efectúa el control y monitorización de parámetros relacionados con el motor que controla el soporte del robot, mediante el empleo del protocolo de comunicaciones CANopen.
- Se ha integrado el desarrollo en una interfaz gráfica para facilitar el uso de la aplicación por parte de los usuarios.
- Se han realizado pruebas y ensayos para verificar el correcto funcionamiento de los elementos software y hardware.

Las pruebas se han realizado sobre una maqueta reducida, con el fin de facilitar el traslado del sistema completo y para evitar el deterioro del soporte del robot en la fase de pruebas. Dicha maqueta consta de un motor y un encoder de tamaño muy reducido. A pesar de este motivo, el diseño realizado puede ser exportado fácilmente al soporte del robot original.

5.2 Trabajos futuros

Las mejoras que se pueden llevar a cabo serían las siguientes:

- A partir de la aplicación creada en este Proyecto Fin de Carrea, desarrollar una arquitectura de control en la que se integren el motor del soporte del robot con el resto de motores presentes en la silla.
- Realizar una interfaz grafica de mayor complejidad, en la que se pueda monitorizar mayor número de parámetros del motor.
- Mejorar el software desarrollado, incluyendo funciones que depuren posibles fallos que puedan producirse durante el funcionamiento de la aplicación.
- Diseñar un sistema de control que regule la inclinación del respaldo de la silla de ruedas, a partir de las funciones desarrolladas en este trabajo.
- Añadir al software paradas de emergencia en el caso de mal funcionamiento u otros motivos.

Bibliografía

[1] **PFC:** Desarrollo de una aplicación para el control simultaneo de motores y generación de trayectorias para el robot humanoide RH-2. Autor: Raúl Morales Tejero. Director: Santiago Martínez de la Casa Díaz.

[2] **PFM:** Diseño e implementación de una arquitectura para el control del robot humanoide RH-2. Autor: José Álvarez Paramio. Director: Santiago Martínez de la Casa Díaz.

[3] **EasyMotion Studio y EasySetup:** Technosoft Product description. http://www.technosoftmotion.com/products/OEM_PROD_EasyMotion.htm

[4] **EEPROM Programmer:** EEPROM Programmer help. http://www.technosoftmotion.com/ESM-um-html/index.html?help_eeprom_programmer.htm

[5] **CAN:** Control Area Network. http://www.can-cia.org/

[6] **CANopen Programming Technosoft:** user manual.

[7] **ASIBOT:** Portable Assistive Robot. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5959995

[8] **Maxon Motor:** products. http://www.maxonmotor.es/maxon/view/content/index

[9] **ISCM8005:** driver Technosoft. http://www.technosoftmotion.com/products/OEM_PROD_ISCM4805.htm

[10] **HICO.CAN-miniPCI:** Inteligent CAN-bus interface for Mini-PCI. http://emtrion.de/hicocan_minipci_en.php

[11] **RoBoard RB-100.**
http://www.roboard.com/RB-100.htm

[12] **ISCM4805/ISCM8005; Versión 1.3; Inteligent Servo Drive for Step, DC, Brushless DC and AC Motors:** Technical Reference.

[13] **Motion Programming using EasyMotion Studio:** user manual.

[14] **Linux CAN driver manual. HICO.CAN-miniPCI, PCI-104 and PC/104+ boards**.